# 4.2 对Direct3D进行初始化

下面的各小节将讲解如何初始化Direct3D。我们将Direct3D的初始化过程分为如下几个步骤：

1. 使用**D3D11CreateDevice**方法创建**ID3D11Device**和**ID3D11DeviceContext**。
2. 使用**ID3D11Device::CheckMultisampleQualityLevels**方法检测设备支持的4X多重采样质量等级。
3. 填充一个**IDXGI\_SWAP\_CHAIN\_DESC**结构体，该结构体描述了所要创建的交换链的特性。
4. 查询**IDXGIFactory**实例，这个实例用于创建设备和一个**IDXGISwapChain**实例。
5. 为交换链的后台缓冲区创建一个渲染目标视图。
6. 创建深度/模板缓冲区以及相关的深度/模板视图。
7. 将渲染目标视图和深度/模板视图绑定到渲染管线的输出合并阶段，使它们可以被Direct3D使用。
8. 设置视口。

## 4.2.1 创建设备（Device）和上下文（Context）

要初始化Direct3D，首先需要创建Direct3D 11设备（**ID3D11Device**）和上下文（**ID3D11DeviceContext**）。它们是是最重要的Direct3D接口，可以被看成是物理图形设备硬件的软控制器；也就是说，我们可以通过该接口与硬件进行交互，命令硬件完成一些工作（比如：在显存中分配资源、清空后台缓冲区、将资源绑定到各种管线阶段、绘制几何体）。具体而言：

1. **ID3D11Device**接口用于检测显示适配器功能和分配资源。
2. **ID3D11DeviceContext**接口用于设置管线状态、将资源绑定到图形管线和生成渲染命令。

设备和上下文可用如下函数创建：

HRESULT D3D11CreateDevice (

IDXGIAdapter \*pAdapter,

D3D\_DRIVER\_TYPE DriverType,

HMODULE Software ,

UINT Flags ,

CONST D3D\_FEATURE\_LEVEL \*pFeatureLevels ,

UINT FeatureLevels ,

UINT SDKVersion,

ID3D11Device \*\*ppDevice ,

D3D\_FEATURE\_LEVE L \*pFeatureLevel,

ID3D11DeviceContext \*\*ppImmediateContext

);

1．**pAdapter**：指定要为哪个物理显卡创建设备对象。当该参数设为空值时，表示使用主显卡。在本书的示例程序中，我们只使用主显卡。

2．**DriverType**：一般来讲，该参数总是指定为**D3D\_DRIVER\_TYPE\_HARDWARE**，表示使用3D硬件来加快渲染速度。但是，也可以有两个其他选择：

**D3D\_DRIVER\_TYPE\_REFERENCE**：创建所谓的引用设备（reference device）。引用设备是Direct3D的软件实现，它具有Direct3D的所有功能（只是运行速度非常慢，因为所有的功能都是用软件来实现的）。引用设备随DirectX SDK一起安装，只用于程序员，而不应该用于程序发布。使用引用设备有两个原因：

* 测试硬件不支持的代码；例如，在一块不支持Direct3D 11的显卡上测试一段Direct3D 11的代码。
* 测试驱动程序缺陷。当代码能在引用设备上正常运行，而不能在硬件上正常工作时，说明硬件的驱动程序可能存在缺陷。

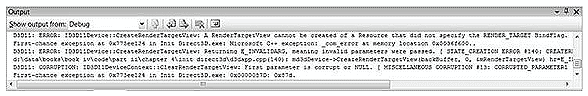
**D3D\_DRIVER\_TYPE\_SOFTWARE**：创建一个用于模拟3D硬件的软件驱动器。要使用软件驱动器，你必须自己创建一个，或使用第三方的软件驱动器。与下面要说的WARP驱动器不同，Direct3D不提供软件驱动器。

**D3D\_DRIVER\_TYPE\_WARP**：创建一个高性能的Direct3D 10.1软件驱动器。WARP代表Windows Advanced Rasterizati on Platform。因为WARP不支持Direct3D 11，因此我们对它不感兴趣。

3．**Software**：用于支持软件光栅化设备（software rasterizer）。我们总是将该参数设为空值，因为我们使用硬件进行渲染。如果读者想要使用这一功能，那么就必须先安装一个软件光栅化设备。

4．**Flags**：可选的设备创建标志值。当以release模式生成程序时，该参数通常设为0（无附加标志值）；当以debug模式生成程序时，该参数应设为：

**D3D11\_CREATE\_DEVICE\_DEBUG**：用以激活调试层。当指定调试标志值后，Direct3D会向VC++的输出窗口发送调试信息；图4.6展示了输出错误信息的一个例子。



**图4.6 Direct3D 11调试输出的一个例子。**

5．**pFeatureLevels**：**D3D\_FEATURE\_LEVEL**数组，元素的顺序表示要特征等级（见§4.1.9）的测试顺序。将这个参数设置为null表示选择可支持的最高等级。

6．**FeatureLevels**：**pFeatureLevels**数组中的元素**D3D\_FEATURE\_LEVELs**的数量，若**pFeatureLevels**设置为null，则这个值为0。

7．**SDKVersion**：始终设为**D3D11\_SDK\_VERSION**。

8．**ppDevice**：返回创建后的设备对象。

9．**pFeatureLevel**：返回**pFeatureLevels**数组中第一个支持的特征等级（如果**pFeatureLevels** 为null，则返回可支持的最高等级）。

10．**ppImmediateContext**：返回创建后的设备上下文。

下面是调用该函数的一个示例：

UINT createDeviceFlags = 0;

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

createDeviceFlags |= D3D11\_CREATE\_DEVICE\_DEBUG;

#endif

D3D\_FEATURE\_LEVEL featureLevel;

ID3D11Device \* md3dDevice;

ID3D11Device Context\* md3dImmediate Context;

HRESULT hr = D3D11CreateDevice(

0, // 默认显示适配器

D3D\_DRIVER\_TYPE\_HARDWARE ,

0, // 不使用软件设备

createDeviceFlags ,

0, 0, // 默认的特征等级数组

D3D11\_SDK\_VERSION,

& md3dDevice ,

& featureLevel,

& md3dImmediateContext);

if(FAILED(hr) )

{

MessageBox(0, L"D3D11CreateDevice Failed.", 0, 0);

return false ;

}

if(featureLevel != D3D\_FEATURE\_LEVEL\_11\_0)

{

MessageBox(0, L"Direct3D FeatureLevel 11 unsupported.", 0, 0);

return false;

}

从上面的代码可以看到我们使用的是立即执行上下文（immediate context）：

**ID3D11DeviceContext\* md3dImmediateContext;**

还有一种上下文叫做延迟执行上下文（**ID3D11Device::CreateDeferredContext**）。该上下文主要用于Direct3D 11支持的多线程程序。多线程编程是一个高级话题，本书并不会介绍，但下面介绍一点基本概念：

1．在主线程中使用立即执行上下文。

2．在工作线程总使用延迟执行上下文。

（a）每个工作线程可以将图形指令记录在一个命令列表（**ID3D11CommandList**）中。

（b）随后，每个工作线程中的命令列表可以在主渲染线程中加以执行。

在多核系统中，可并行处理命令列表中的指令，这样可以缩短编译复杂图形所需的时间。

## 4.2.2 检测4X多重采样质量支持

创建了设备后，我们就可以检查4X多重采样质量等级了。所有支持Direct3D 11的设备都支持所有渲染目标格式的4X MSAA（支持的质量等级可能并不相同）。

UINT m4xMsaaQuality;

HR(md3dDevice ->CheckMultisampleQualityLevels(

DXGI\_FORMAT\_R8G8B8A8\_UNORM, 4, & m4xMsaaQuality));

assert(m4xMsaaQuality > 0);

因为4X MSAA总是被支持的，所以返回的质量等级总是大于0。

## 4.2.3 描述交换链

下一步是创建交换链，首先需要填充一个**DXGI\_SWAP\_CHAIN\_DESC**结构体来描述我们将要创建的交换链的特性。该结构体的定义如下：

typedef struct DXGI\_SWAP\_CHAIN\_DESC {

DXGI\_MODE\_DESC BufferDesc;

DXGI\_SAMPLE\_DESC SampleDesc;

DXGI\_USAGE BufferUsage;

UINT BufferCount;

HWND OutputWindow;

BOOL Windowed;

DXGI\_SWAP\_EFFECT SwapEffect;

UINT Flags;

} DXGI\_SWAP\_CHAIN\_DESC;

**DXGI\_MODE\_DESC**类型是另一个结构体，其定义如下：

typedef struct DXGI\_MODE\_DESC

{

UINT Width; // 后台缓冲区宽度

UINT Height; // 后台缓冲区高度

DXGI\_RATIONAL RefreshRate; // 显示刷新率

DXGI\_FORMAT Format; // 后台缓冲区像素格式

DXGI\_MODE\_SCANLINE\_ORDER ScanlineOrdering;// display scanline mode

DXGI\_MODE\_SCALING Scaling; // display scaling mode

} DXGI\_MODE\_DESC;

注意：在下面的数据成员描述中，我们只涵盖了一些常用的标志值和选项，它们对于初学者来说非常重要。对于其他标志值和选项的描述，请参阅SDK文档。

1. **BufferDesc**：该结构体描述了我们所要创建的后台缓冲区的属性。我们主要关注的属性有：宽度、高度和像素格式；其他属性的详情请参阅SDK文档。
2. **SampleDesc**：多重采样数量和质量级别（参阅4.1.8节）。
3. **BufferUsage**：设为**DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT**，因为我们要将场景渲染到后台缓冲区（即，将它用作渲染目标）。
4. **BufferCount**：交换链中的后台缓冲区数量；我们一般只用一个后台缓冲区来实现双缓存。当然，你也可以使用两个后台缓冲区来实现三缓存。
5. **OutputWindow**：我们将要渲染到的窗口的句柄。
6. **Windowed**：当设为**true**时，程序以窗口模式运行；当设为**false**时，程序以全屏（full-screen）模式运行。
7. **SwapEffect**：设为**DXGI\_SWAP\_EFFECT\_DISCARD**，让显卡驱动程序选择最高效的显示模式。
8. **Flags**：可选的标志值。如果设为**DXGI\_SWAP\_CHAIN\_FLAG\_ALLOW\_MODE\_SWITCH**，那么当应用程序切换到全屏模式时，Direct3D会自动选择与当前的后台缓冲区设置最匹配的显示模式。如果未指定该标志值，那么当应用程序切换到全屏模式时，Direct3D会使用当前的桌面显示模式。我们在示例框架中没有使用该标志值，因为对于我们的演示程序来说，在全屏模式下使用当前的桌面显示模式可以得到很好的效果。

下面是在我们的示例框架中填充**DXGI\_SWAP\_CHAIN\_DESC**结构体的代码：

DXGI\_SWAP\_CHAIN\_DESC sd;

sd.BufferDesc.Width = mClientWidth; // 使用窗口客户区宽度

sd.BufferDesc.Height = mClientHeight;

sd.BufferDesc.RefreshRate.Numerator = 60;

sd.BufferDesc.RefreshRate.Denominator = 1;

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

sd.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED;

sd.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED;

// 是否使用4X MSAA?

if(mEnable4xMsaa)

{

sd.SampleDesc.Count = 4;

// m4xMsaaQuality是通过CheckMultisampleQualityLevels()方法获得的

sd.SampleDesc.Quality = m4xMsaaQuality-1;

}

// NoMSAA

else

{

sd.SampleDesc.Count = 1;

sd.SampleDesc.Quality = 0;

}

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

sd.BufferCount = 1;

sd.OutputWindow = mhMainWnd;

sd.Windowed = true;

sd.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD;

sd.Flags = 0;

**注意**：如果你想在运行时改变多重采样的设置，那么必须销毁然后重新创建交换链。

**注意**：因为大多数显示器不支持超过24位以上的颜色，再多的颜色也是浪费，所以我们将后台缓冲区的像素格式设置为**DXGI\_FORMAT\_R8G8B8A8\_UNORM**（红、绿、蓝、alpha各8位）。额外的8位alpha并不会输出在显示器上，但在后台缓冲区中可以用于特定的用途。

## 4.2.4 创建交换链

交换链（**IDXGISwapChain**）是通过**IDXGIFactory**实例的**IDXGIFactory::CreateSwapChain**方法创建的：

HRESULT IDXGIFactory::CreateSwapChain(

IUnknown \*pDevice , // 指向ID3D11Device的指针

DXGI\_SWAP\_CHAIN\_DESC \*pDesc, // 指向一个交换链描述的指针

IDXGISwapChain \*\*ppSwapChain); // 返回创建后的交换链

我们可以通过**CreateDXGIFactory**（需要链接dxgi.lib）获取指向一个**IDXGIFactory**实例的指针。但是使用这种方法获取**IDXGIFactory**实例，并调用**IDXGIFactory::CreateSwapChain**方法后，会出现如下的错误信息：

DXGI Warning: IDXGIFactory::CreateSwapChain: This function is being called with a device from a different IDXGIFactory.

要修复这个错误，我们需要使用创建设备的那个**IDXGIFactory**实例，要获得这个实例，必须使用下面的COM查询（具体解释可参见**IDXGIFactory**的文档）：

IDXGIDevice \* dxgiDevice = 0;

HR(md3dDevice ->QueryInterface(\_\_uuidof(IDXGIDevice),

(void\*\*)&dxgiDevice ));

IDXGIAdapter\* dxgiAdapter = 0;

HR(dxgiDevice ->GetParent(\_\_uuidof(IDXGIAdapter),

(void\*\*))&dxgiAdapte r ));

// 获得IDXGIFactory 接口

IDXGIFactory\* dxgiFactory = 0;

HR(dxgiAdapter->GetParent(\_\_uuid of(IDXGIFactory),

(void\*\*))&dxgiFactor y));

// 现在，创建交换链

IDXGISwapChain\* mSwapChain;

HR(dxgiFactory->CreateSwapChain(md3dDevice, &sd , &mSw ap Chain);

// 释放COM接口

ReleaseCOM (dxgiDevice ;

ReleaseCOM (dxgiAdapter);

ReleaseCOM (dxgiFactory);

DXGI（DirectX Graphics Inf rastructure）是独立于Direct3D的API，用于处理与图形关联的东西，例如交换链等。DXGI与Direct3D分离的目的在于其他图形API（例如Direct2D）也需要交换链、图形硬件枚举、在窗口和全屏模式之间切换，通过这种设计，多个图形API都能使用DXGI API。

**补充**：你也可以使用**D3D11CreateDeviceAndSwapChain**方法同时创建设备、设备上下文和交换链，详情请见[Direct3D 11 教程1：Direct3D 11基础](http://shiba.hpe.sh.cn/jiaoyanzu/WULI/showArticle.aspx?articleId=923&classId=5)。

## 4.2.5 创建渲染目标视图

如4.1.6节所述，资源不能被直接绑定到一个管线阶段；我们必须为资源创建资源视图，然后把资源视图绑定到不同的管线阶段。尤其是在把后台缓冲区绑定到管线的输出合并器阶段时（使Direct3D可以在后台缓冲区上执行渲染工作），我们必须为后台缓冲区创建一个**渲染目标视图（render target view）**。下面的代码说明了一实现过程：

ID3D11RenderTargetView\* mRenderTargetView;

ID3D11Texture2D\* backBuffer;

mSwapChain->GetBuffer(0,\_\_uuidof(ID3D11Texture2D),

reinterpret\_cast<void\*\*>(&backBuffer));

md3dDevice->CreateRenderTargetView(backBuffer, 0, &mRenderTargetView);

ReleaseCOM(backBuffer);

1. **IDXGISwapChain::GetBuffer**方法用于获取一个交换链的后台缓冲区指针。该方法的第一个参数表示所要获取的后台缓冲区的索引值（由于后台缓冲区的数量可以大于1，所以这里必须指定索引值）。在我们的演示程序中，我们只使用一个后台缓冲区，所以该索引值设为0。第二个参数是缓冲区的接口类型，它通常是一个2D纹理（**ID3D11Texture2D**）。第三个参数返回指向后台缓冲区的指针。
2. 我们使用**ID3D11Device::CreateRenderTargetView**方法创建渲染目标视图。第一个参数指定了将要作为渲染目标的资源，在上面的例子中，渲染目标是后台缓冲区（即，我们为后台缓冲区创建了一个渲染目标视图）。第二个参数是一个指向**D3D11\_RENDER\_TARGET\_VIEW\_DESC**结构体的指针，该结构体描述了资源中的元素的数据类型。如果在创建资源时使用的是某种强类型格式（即，非弱类型格式），则该参数可以为空，表示以资源的第一个mipmap层次（后台缓冲区也只有一个mipmap层次）作为视图格式。第三个参数通过指针返回了创建后的渲染目标视图对象。
3. 每调用一次**IDXGISwapChain::GetBuffer**方法，后台缓冲区的COM引用计数就会向上递增一次，这便是我们在代码片段的结尾处释放它（**ReleaseCOM**）的原因。

## 4.2.6 创建深度/模板缓冲区及其视图

我们现在需要创建深度/模板缓冲区。如4.1.5节所述，深度缓冲区只是一个存储深度信息的2D纹理（如果使用模板，则模板信息也在该缓冲区中）。要创建纹理，我们必须填充一个**D3D11\_TEXTURE2D\_DESC**结构体来描述所要创建的纹理，然后再调用**ID3D11Device::CreateTexture2D**方法。该结构体的定义如下：

typedef struct D3D11\_TEXTURE2D\_DESC {

UINT Width;

UINT Height;

UINT MipLevels;

UINT ArraySize;

DXGI\_FORMAT Format;

DXGI\_SAMPLE\_DESC SampleDesc;

D3D10\_USAGE Usage;

UINT BindFlags;

UINT CPUAccessFlags;

UINT MiscFlags;

} D3D11\_TEXTURE2D\_DESC;

1. **Width**：纹理的宽度，单位为纹理元素（texel）。
2. **Height**：纹理的高度，单位为纹理元素（texel）。
3. **MipLevels**：多级渐近纹理层（mipmap level）的数量。多级渐近纹理将在后面的章节“纹理”中进行讲解。对于深度/模板缓冲区来说，我们的纹理只需要一个多级渐近纹理层。
4. **ArraySize**：在纹理数组中的纹理数量。对于深度/模板缓冲区来说，我们只需要一个纹理。
5. **Format**：一个**DXGI\_FORMAT**枚举类型成员，它指定了纹理元素的格式。对于深度/模板缓冲区来说，它必须是4.1.5节列出的格式之一。
6. **SampleDesc**：多重采样数量和质量级别；请参阅4.1.7和4.1.8节。
7. **Usage**：表示纹理用途的**D3D11\_USAGE**枚举类型成员。有4个可选值：

* **D3D11\_USAGE\_DEFAULT**：表示GPU（graphics processing unit，图形处理器）会对资源执行读写操作。CPU不能读写这种资源。对于深度/模板缓冲区，我们使用**D3D11\_USAGE\_DEFAULT**标志值，因为GPU会执行所有读写深度/模板缓冲区的操作。
* **D3D10\_USAGE\_IMMUTABLE**：表示在创建资源后，资源中的内容不会改变。这样可以获得一些内部优化，因为GPU会以只读方式访问这种资源。除了在创建资源时CPU会写入初始化数据外，其他任何时候CPU都不会对这种资源执行任何读写操作。
* **D3D10\_USAGE\_DYNAMIC**：表示应用程序（CPU）会频繁更新资源中的数据内容（例如，每帧更新一次）。GPU可以从这种资源中读取数据，而CPU可以向这种资源中写入数据。
* **D3D10\_USAGE\_STAGING**：表示应用程序（CPU）会读取该资源的一个副本（即，该资源支持从显存到系统内存的数据复制操作）。

1. **BindFlags**：指定该资源将会绑定到管线的哪个阶段。对于深度/模板缓冲区，该参数应设为**D3D11\_BIND\_DEPTH\_STENCIL**。其他可用于纹理的绑定标志值还有：

* **D3D11\_BIND\_RENDER\_TARGET**：将纹理作为一个渲染目标绑定到管线上。
* **D3D11\_BIND\_SHADER\_RESOURCE**：将纹理作为一个着色器资源绑定到管线上。

1. **CPUAccessFlags**：指定CPU对资源的访问权限。如果CPU需要向资源写入数据，则应指定**D3D11\_CPU\_ACCESS\_WRITE**。具有写访问权限的资源的Usage参数应设为**D3D11\_USAGE\_DYNAMIC**或**D3D11\_USAGE\_STAGING**。如果CPU需要从资源读取数据，则应指定**D3D11\_CPU\_ACCESS\_READ**。具有读访问权限的资源的Usage参数应设为**D3D11\_USAGE\_STAGING**。对于深度/模板缓冲区来说，只有GPU会执行读写操作；所以，我们将该参数设为0，因为CPU不会在深度/模板缓冲区上执行读写操作。
2. **MiscFlags**：可选的标志值，与深度/模板缓冲区无关，所以设为0。

**注意**：推荐避免使用**D3D11\_USAGE\_DYNAMIC**和**D3D11\_USAGE\_STAGING**，因为有性能损失。要获得最佳性能，我们应创建所有的资源并将它们上传到GPU并保留其上，只有GPU在读取或写入这些资源。但是，在某些程序中必须有CPU的参与，因此这些标志无法避免，但你应该将这些标志的使用减到最小。

在本书中，我们会看到以各种不同选项来创建资源的例子；例如，使用不同的Usage标志值、绑定标志值和CPU访问权限标志值。但就目前来说，我们只需要关心那些与创建深度/模板缓冲区有关的标志值即可，其他选项可以以后再说。

另外，在使用深度/模板缓冲区之前，我们必须为它创建一个绑定到管线上的深度/模板视图。过程与创建渲染目标视图的过程相似。下面的代码示范了如何创建深度/模板纹理以及与它对应的深度/模板视图：

D3D11\_TEXTURE2D\_DESC depthStencilDesc;

depthStencilDesc.Width = mClientWidth;

depthStencilDesc.Height = mClientHeight;

depthStencilDesc.MipLevels = 1;

depthStencilDesc.ArraySize = 1;

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

// 使用4X MSAA?——必须与交换链的MSAA的值匹配

if( mEnable4xMsaa)

{

depthStencilDesc.SampleDesc.Count = 4;

depthStencilDesc.SampleDesc.Quality = m4xMsaaQuality-1;

}

// 不使用MSAA

else

{

depthStencilDesc.SampleDesc.Count = 1;

depthStencilDesc.SampleDesc.Quality = 0;

}

depthStencilDesc.Usage = D3D10\_USAGE\_DEFAULT;

depthStencilDesc.BindFlags = D3D10\_BIND\_DEPTH\_STENCIL;

depthStencilDesc.CPUAccessFlags = 0;

depthStencilDesc.MiscFlags = 0;

ID3D10Texture2D\* mDepthStencilBuffer;

ID3D10DepthStencilView\* mDepthStencilView;

HR(md3dDevice->CreateTexture2D(

&depthStencilDesc, 0, &mDepthStencilBuffer));

HR(md3dDevice->CreateDepthStencilView(

mDepthStencilBuffer, 0, &mDepthStencilView));

**CreateTexture2D**的第二个参数是一个指向初始化数据的指针，这些初始化数据用来填充纹理。不过，由于个纹理被用作深度/模板缓冲区，所以我们不需要为它填充任何初始化数据。当执行深度缓存和模板操作时，Direct3D会自动向深度/模板缓冲区写入数据。所以，我们在这里将第二个参数指定为空值。

**CreateDepthStencilView**的第二个参数是一个指向**D3D11\_DEPTH\_STENCIL\_VIEW\_DESC**的指针。这个结构体描述了资源中这个元素数据类型（格式）。如果资源是一个有类型的格式（非typeless），这个参数可以为空值，表示创建一个资源的第一个mipmap等级的视图（深度/模板缓冲也只能使用一个 mipmap等级）。因为我们指定了深度/模板缓冲的格式，所以将这个参数设置为空值。

## 4.2.7 将视图绑定到输出合并器阶段

现在我们已经为后台缓冲区和深度缓冲区创建了视图，就可以将些视图绑定到管线的输出**合并器阶段（output merger stage）**，使些资源成为管线的渲染目标和深度/模板缓冲区：

md3dImmediateContext->OMSetRenderTargets(

1,&mRenderTargetView,mDepthStencilView);

第一个参数是我们将要绑定的渲染目标的数量；我们在这里仅绑定了一个渲染目标，不过该参数可以为着色器同时绑定多个渲染目标（是一项高级技术）。第二个参数是我们将要绑定的渲染目标视图数组中的第一个元素的指针。第三个参数是将要绑定到管线的深度/模板视图。

**注意**：我们可以设置一组渲染目标视图，但是只能设置一个深度/模板视图。使用多个渲染目标是一项高级技术，会在本书的第三部分加以介绍。

## 4.2.8 设置视口

通常我们会把3D场景渲染到整个后台缓冲区上。不过，有时我们只希望把3D场景渲染到后台缓冲区的一个子矩形区域中，如图4.7所示。

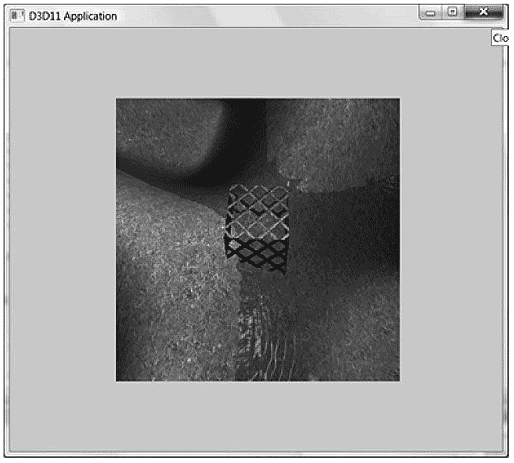


图4.7：通过修改视口，我们可以把3D场景渲染到后台缓冲区的一个子矩形区域中。随后，后台缓冲区中的渲染结果会被呈现到窗口客户区上。

我们将后台缓冲区的子矩形区域称为**视口（viewport）**，它由如下结构体描述：

typedef struct D3D11\_VIEWPORT {

FLOAT TopLeftX;

FLOAT TopLeftY;

FLOAT Width;

FLOAT Height;

FLOAT MinDepth;

FLOAT MaxDepth;

} D3D11\_VIEWPORT;

前4个数据成员定义了相对于窗口客户区的视口矩形范围。MinDepth 成员表示深度缓冲区的最小值，**MaxDepth**表示深度缓冲区的最大值。Direct3D使用的深度缓冲区取值范围是0到1，除非你想要得到一些特殊效果，否则应将**MinDepth**和**MaxDepth**分别设为0和1。

在填充了**D3D11\_VIEWPORT**结构体之后，我们可以使用**ID3D11Device::RSSetViewports**方法设置Direct3D的视口。下面的例子创建和设置了一个视口，该视口与整个后台缓冲区的大小相同：

D3D11\_VIEWPORT vp;

vp.TopLeftX = 0;

vp.TopLeftY = 0;

vp.Width = static\_cast<float>(mClientWidth);

vp.Height = static\_cast<float>(mClientHeight);

vp.MinDepth = 0.0f;

vp.MaxDepth = 1.0f;

md3dImmediateContext-->RSSetViewports(1, &vp);

第一个参数是绑定的视图的数量（可以使用超过1的数量用于高级的效果），第二个参数指向一个viewports的数组。

例如，你可以使用视口来实现双人游戏模式中的分屏效果。创建两个视口，各占屏幕的一半，一个居左，另一个居右。然后在左视口中以第一个玩家的视角渲染3D场景，在右视口中以第二个玩家的视角渲染3D场景。你也可以使用视口只绘制到屏幕的一个子矩形中，而在其他区域保留诸如按钮、列表框之类的UI控件。