# 4.4 演示程序框架

本书中的演示程序均使用d3dUtil.h、d3dApp.h、d3dApp.cpp文件中的代码，这些文件可以从本书网站下载。由于本书的第Ⅱ部分和第Ⅲ部分的所有演示程序都会用到些常用文件，所以我们把些文件保存在了Common目录下，使些文件被所有的工程共享，避免多次复制文件。d3dUtil.h文件包含了一些有用的工具代码，d3dApp.h和d3dApp.cpp文件包含了Direct3D应用程序类的核心代码。我们希望读者在阅读本章之后，仔细研究一下些文件，因为我们不会涵盖些文件中的每一行代码（例如，我们不会讲解如何创建一个Windows窗口，因为基本的Win32编程是阅读本书的先决条件）。该框架的目标是隐藏窗口的创建代码和Direct3D的初始化代码；通过隐藏些代码，我们可以在设计演示程序时减少注意力的分散，把注意力集中在示例程序所要表达的特定细节上。

## 4.4.1 D3DApp

**D3DApp**是所有Direct3D应用程序类的基类，它提供了用于创建主应用程序窗口、运行应用程序消息循环、处理窗口消息和初始化Direct3D的函数。另外，这个类还定义了一些框架函数。所有的Direct3D 应用程序类都继承于**D3DApp**类，重载它的virtual框架函数，并创建一个**D3DApp**派生类的单例对象。**D3DApp**类的定义如下：

#ifndef D3DAPP\_H

#define D3DAPP\_H

#include "d3dUtil.h"

#include "GameTimer.h"

#include <string>

class D3DApp

{

public:

 D3DApp(HINSTANCE hInstance);

 virtual ~D3DApp();

 HINSTANCE AppInst()const;

 HWND MainWnd()const;

 float AspectRatio()const;

 int Run();

 // 框架方法。派生类需要重载这些方法实现所需的功能。

 virtual bool Init();

 virtual void OnResize();

 virtual void UpdateScene(float dt)=0;

 virtual void DrawScene()=0;

 virtual LRESULT MsgProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

 // 处理鼠标输入事件的便捷重载函数

 virtual void OnMouseDown(WPARAM btnState, int x, int y){ }

 virtual void OnMouseUp(WPARAM btnState, int x, int y) { }

 virtual void OnMouseMove(WPARAM btnState, int x, int y){ }

protected:

 bool InitMainWindow();

 bool InitDirect3D();

 void CalculateFrameStats();

protected:

 HINSTANCE mhAppInst; // 应用程序实例句柄

 HWND mhMainWnd; // 主窗口句柄

 bool mAppPaused; // 程序是否处在暂停状态

 bool mMinimized; // 程序是否最小化

 bool mMaximized; // 程序是否最大化

 bool mResizing; // 程序是否处在改变大小的状态

 UINT m4xMsaaQuality;// 4X MSAA质量等级

 // 用于记录"delta-time"和游戏时间(§4.3)

 GameTimer mTimer;

 // D3D11设备(§4.2.1)，交换链(§4.2.4)，用于深度/模板缓存的2D纹理(§4.2.6)，

 // 渲染目标(§4.2.5)和深度/模板视图(§4.2.6)，和视口(§4.2.8)。

 ID3D11Device\* md3dDevice;

 ID3D11DeviceContext\* md3dImmediateContext;

 IDXGISwapChain\* mSwapChain;

 ID3D11Texture2D\* mDepthStencilBuffer;

 ID3D11RenderTargetView\* mRenderTargetView;

 ID3D11DepthStencilView\* mDepthStencilView;

 D3D11\_VIEWPORT mScreenViewport;

 // 下面的变量是在D3DApp构造函数中设置的。但是，你可以在派生类中重写这些值。

 // 窗口标题。D3DApp的默认标题是"D3D11 Application"。

 std::wstring mMainWndCaption;

 // Hardware device还是reference device？D3DApp默认使用D3D\_DRIVER\_TYPE\_HARDWARE。

 D3D\_DRIVER\_TYPE md3dDriverType;

 // 窗口的初始大小。D3DApp默认为800x600。注意，当窗口大小在运行阶段改变时，这些值也会随之改变。

 int mClientWidth;

 int mClientHeight;

 // 设置为true则使用4XMSAA(§4.1.8)，默认为false。

 bool mEnable4xMsaa;

};

#endif // D3DAPP\_H

在上面的代码中，我们使用注释描述了一些数据成员的含义；这些方法将在随后的几节中讨论。

## 4.4.2 非框架方法

**1．D3DApp**：构造函数，将数据成员简单地初始化为默认值。

**2．~D3DApp**：析构函数，释放**D3DApp**获取的COM接口。

**3．AppInst**：简单的取值函数，返回应用程序实例句柄的一个副本。

**4．MainWnd**：简单的取值函数，返回主窗口句柄的一个副本。

**5．AspectRatio**：后台缓存区的长宽比，这个比值会在下一章中用到，可以通过下面的代码获得：

float D3DApp::AspectRatio() const

{

 return static\_cast<float>(mClientWidth)/mClientHeight;

}

**6．Run**：该方法封装了应用程序消息循环。它使用Win32 **PeekMessage**函数，当没有消息时，它让应用程序处理我们的游戏逻辑。该函数的实现代码请参见4.3.3节。

**7．InitMainWindow**：初始化主应用程序窗口；我们假定读者已经具备了基本的Win32编程知识，知道如何初始化一个Windows窗口。

**8．InitDirect3D**：通过4.2节描述的各个步骤初始化Direct3D。

**9．CalculateFrameStats**：计算每秒的平均帧数和每帧的平均时间（单位为毫秒），这个方法的实现在4.4.4节中介绍。

## 4.4.3 框架方法

在本书的每个演示程序中，我们都会重载**D3DApp**的5个virtual函数。这5个函数用于实现特定示例中的代码细节。**D3DApp**类实现的这种结构可以将所有的初始化代码、消息处理代码和其他代码安排得井井有条，使派生类专注于实现演示程序的特定代码。下面是对这些框架方法的描述：

**1．Init**：该方法包含应用程序的初始化代码，比如分配资源、初始化对象和设置灯光。该方法在**D3DApp**的实现中包含**InitMainWindow**和**InitDirect3D**方法的调用语句；所以，当在派生类中重载该方法时，应首先调用该方法的D3DApp版本，就像下面这样：

void TestApp::Init()

{

 if(!D3DApp::Init())

 return false;

 /\* 剩下的初始化代码从这里开始 \*/

}

为你的后续初始化代码提供一个可用的**ID3D11Device**设备对象。（通常在获取 Direct3D资源时都要传递一个有效的**ID3D11Device**设备对象。）

**2．OnResize**：该方法在**D3DApp::MsgProc**收到**WM\_SIZE**消息时调用。当窗口的尺寸改变时，一些与客户区大小相关的 Direct3D属性也需要改变。尤其是需要重新创建后台缓冲区和深度/模板缓冲区，使它们与窗口客户区的大小一致。后台缓冲区的大小可以通过调用**IDXGISwapChain::ResizeBuffers**方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。**OnResize**方法在**D3DApp**的实现中包含了调整后台缓冲区和深度/模板缓冲区的代码；详情请直接参见源代码。除缓冲区外，依赖于客户区大小的其他属性（例如，投影矩阵）也必须重新创建。我们把该方法作为框架的一部分是因为当窗口大小改变时，客户代码可能需要执行一些它自己的逻辑。

**3．UpdateScene**：该抽象方法每帧都会调用，用于随着时间更新3D应用程序（例如，实现动画和碰撞检测、检查用户输入、计算每秒帧数等等）。

**4．DrawScene**：该抽象方法每帧都会调用，用于将3D场景的当前帧绘制到后台缓冲区。当绘制当前帧时，我们调用了**IDXGISwapChain::Present**方法将后台缓冲区的内容呈现在屏幕上。

**5．MsgProc**：该方法是主应用程序窗口的消息处理函数。通常，当你只需重载该方法，就可以处理未由**D3DApp::MsgProc**处理（或者没按照你所希望的方式处理）的消息。该方法的**D3DApp**实现版本会在4.4.5节中讲解。如果你重载了这个方法，那么那些你没有处理的消息都会送到**D3DApp::MsgProc**中进行处理。

**注意**：除了上述的五个框架方法之外，为了使用起来更方便，我们还提供了三个虚函数，用于处理鼠标点击、释放和移动的事件。

virtual void OnMouseDown(WPARAM btnState, int x, int y){ }

virtual void OnMouseUp(WPARAM btnState, int x, int y) { }

virtual void OnMouseMove(WPARAM btnState, int x, int y){ }

你可以重载这些方法处理鼠标事件，而用不着重载**MsgProc**方法。这些方法的第一个参数**WPARAM**都是相同的，保存了鼠标按键的状态（例如，哪个鼠标按键被按下），第二、三个参数是光标在客户区域的（x，y）坐标。

## 4.4.4 帧的统计数值

通常游戏和绘图应用程序都要测量每秒的渲染帧数（FPS）。要实现这一工作，我们只需计算在某一特定时间段*t*中处理的总帧数（并存储在中变量*n*中）。然后得到时间段*t*中的平均FPS为*fps*avg＝*n*/*t*。如果我们将*t*设为1，那么*fps*avg＝*n*/1＝*n*。在我们的代码中，我们将*t*设为1，这样可以减少一次除法操作，而且，以1秒为限可以得到一个最恰当的平均值——个时间间隔既不长也不短。计算FPS的代码由**D3Dapp::CalculateFrameStats**方法实现：

void D3DApp::CalculateFrameStats()

{

 // 计算每秒平均帧数的代码，还计算了绘制一帧的平均时间。

 // 这些统计信息会显示在窗口标题栏中。

 static int frameCnt = 0;

 static float timeElapsed = 0.0f;

 frameCnt++;

 // 计算一秒时间内的平均值

 if( (mTimer.TotalTime() - timeElapsed) >= 1.0f )

 {

 float fps = (float)frameCnt; // fps = frameCnt / 1

 float mspf = 1000.0f / fps;

 std::wostringstream outs;

 outs.precision(6);

 outs << mMainWndCaption << L" "

 << L"FPS: " << fps << L" "

 << L"Frame Time: " << mspf << L" (ms)";

 SetWindowText(mhMainWnd, outs.str().c\_str());

 // 为了计算下一个平均值重置一些值。

 frameCnt = 0;

 timeElapsed += 1.0f;

 }

}

为了统计帧数，我们在每帧中都会调用该方法。

除了计算FPS外，上面的代码还计算了处理一帧所花费的平均时间，单位为毫秒：

float mspf = 1000.0f / fps;

**注意**：帧时间与FPS是倒数关系，通过乘以1000ms/ 1s可以将秒转换为毫秒（1秒等于1000毫秒）。

这条语句的含义是：以毫秒为单位计算渲染一帧所花费的时间；是一个与FPS不同的值（虽然个值源于FPS）。实际上，计算帧时间比计算FPS更有用，因为它可以更直观地反映出由于修改场景而产生的渲染时间变化（增加或减少）。另一方面，FPS无法反映出这一变化。而且，[Dunlop03]在他的文章《FPS versus Frame Time》中指出：由于FPS曲线是非线性的，所以使用FPS可能会得到误导性的结果。例如，考虑情景一：假设我们的应用程序以1000FPS的速率运行，每1ms（毫秒）渲染一帧。当帧速率下降到250FPS时，每4ms渲染一帧。现在，再考虑情景二：假设我们的应用程序以的100FPS的速率运行，每10ms渲染一帧。当帧速率下落到大约76.9 FPS时，大约为每13ms渲染一帧。在两个情景中，帧时间都是增加了3毫秒，增加的渲染时间完全相同。但是FPS的读数不够直观。从表面看上，似乎从1000FPS下降到250FPS，要比从100FPS下降到76.9FPS更严重一些。然而，正如我们之前所说，它们实际表示的渲染时间的增长量是相同的。

## 4.4.5 消息处理函数

我们在消息处理函数中实现的代码与整个应用程序框架相比微不足道。通常，我们不会用到许多Win32消息。其实，我们的应用程序的核心代码会在处理器空闲执行（即，当没有窗口消息执行）。不过，有一些重要的消息我们必须处理。因为考虑到篇幅问题，我们不可能在这里列出所有的代码； 我们只能对本例使用的几个消息做以讲解。我们希望读者下载源代码文件，花一些时间熟悉应用程序框架代码，因为它是本书每个示例的基础。

我们处理的第1个消息是**WM\_ACTIVATE**。当应用程序获得焦点或失去焦点时，该消息被发送。我们这样来处理它：

// 当窗口被激活或非激活时会发送WM\_ACTIVATE消息。

// 当非激活时我们会暂停游戏，当激活时则重新开启游戏。

case WM\_ACTIVATE:

 if( LOWORD(wParam) == WA\_INACTIVE )

 {

 mAppPaused = true;

 mTimer.Stop();

 }

 else

 {

 mAppPaused = false;

 mTimer.Start();

 }

 return 0;

可以看到，当应用程序失去焦点时，我们将数据成员**mAppPaused**设为**true**，当应用程序获得焦点时，我们将数据成员**mAppPaused**设为**false**。另外，当应用程序暂停时，计时器停止运行，当应用程序再次激活时，计时器恢复运行。如果回顾4.3.3节中**D3DApp::Run**方法，我们会发现当应用程序暂停时，我们并没有执行应用程序中的更新3D场景的代码，而是将空闲的CPU周期返回给了操作系统；通过这一方式，应用程序不会在处于非活动状态时独占CPU周期。

我们处理的第二个消息是**WM\_SIZE**。该消息在改变窗口大小时发生。我们处理该消息的主要原因是希望后台缓冲区和深度/模板缓冲区的大小与窗口客户区的大小相同（为了不出现图像拉伸）。所以，每次改变窗口大小时，我们希望同改变缓冲区的大小。这一任务由**D3DApp::OnResize**方法实现。如前所述，后台缓冲区的大小可以通过调用**IDXGISwapChain::ResizeBuffers**方法来进行调整。而深度/模板缓冲区必须被销毁，然后根据新的大小重新创建。另外，渲染目标视图和深度/模板视图也必须重新创建。当用户拖动窗口边框时，我们必须格外小心，因为此时会有接连不断的WM\_SIZE消息发出，我们不希望连续地调整缓冲区大小。所以，当用户拖动窗口边框时，我们（除了暂停应用程序外）不应该执行任何代码，等到用户的拖动操作结束之后我们再调整缓冲区的大小。我们通过处理**WM\_EXITSIZEMOVE**消息来完成一工作。该消息在用户释放窗口边框时发送。

// 当用户拖动窗口边框时会发送WM\_EXITSIZEMOVE消息。

case WM\_ENTERSIZEMOVE:

 mAppPaused = true;

 mResizing = true;

 mTimer.Stop();

 return 0;

// 当用户是否窗口边框时会发送WM\_EXITSIZEMOVE消息。

// 然后我们会基于新的窗口大小重置所有图形变量

case WM\_EXITSIZEMOVE:

 mAppPaused = false;

 mResizing = false;

 mTimer.Start();

 OnResize();

 return 0;

最后处理的3个消息的实现过程非常简单，所以我们直接来看代码：

// 窗口被销毁时发送WM\_DESTROY消息

case WM\_DESTROY:

 PostQuitMessage(0);

 return 0;

// 如果使用者按下Alt和一个与菜单项不匹配的字符时，或者在显示弹出式菜单而

// 使用者按下一个与弹出式菜单里的项目不匹配的字符键时。

case WM\_MENUCHAR:

 // 按下alt-enter切换全屏时不发出声响

 return MAKELRESULT(0, MNC\_CLOSE);

// 防止窗口变得过小。

case WM\_GETMINMAXINFO:

 ((MINMAXINFO\*)lParam)->ptMinTrackSize.x = 200;

 ((MINMAXINFO\*)lParam)->ptMinTrackSize.y = 200;

 return 0;

## 4.4.6 全屏模式

我们创建的**IDXGISwapChain**接口可以自动捕获Alt+Enter组合键消息，将应用程序切换到全屏模式（full-screen mode）。在全屏模式下，再次按下Alt+Enter组合键，可以返回到窗口模式。在这两种模式的切换中，应用程序的窗口大小会发生变化，会有一个**WM\_SIZE**消息发送到应用程序的消息队列中；应用程序可以在此时调整后台缓冲区和深度/模板缓冲区的大小，使缓冲区与新的窗口大小匹配。另外，当切换到全屏模式时，窗口样式也会发生改变（即，窗口边框和标题栏会消失）。读者可以使用Visual Studio的Spy++工具查看一下在按下Alt+Enter组合键时由演示程序产生的Windows消息。



**图4.10 第4章示例程序的屏幕截图。**

注意：读者可以回顾一下4.2.3节描述的**DXGI\_SWAP\_CHAIN\_DESC::Flags**标志值。

## 4.4.7 初始化 Direct3D 演示程序

现在，我们已经讨论了应用程序框架的所有内容，下面让我们来使用该框架生成一个小程序。基本上，我们用不着做任何实际工作就可以实现这个程序，因为基类**D3DApp**已经实现了它所需要的大部分功能。读者在这里应该关注是如何编写D3DApp的派生类以及实现框架方法，我们将要在这些框架方法中编写特定的示例代码。本书中的所有程序都使用这一模板。

#include "d3dApp.h"

class InitDirect3DApp : public D3DApp

{

public:

 InitDirect3DApp(HINSTANCE hInstance);

 ~InitDirect3DApp();

 bool Init();

 void OnResize();

 void UpdateScene(float dt);

 void DrawScene();

};

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,

 PSTR cmdLine, int showCmd)

{

 // Enable run-time memory check for debug builds.

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

 \_CrtSetDbgFlag( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF );

#endif

 InitDirect3DApp theApp(hInstance);

 if( !theApp.Init() )

 return 0;

 return theApp.Run();

}

InitDirect3DApp::InitDirect3DApp(HINSTANCE hInstance)

: D3DApp(hInstance)

{

}

InitDirect3DApp::~InitDirect3DApp()

{

}

bool InitDirect3DApp::Init()

{

 if(!D3DApp::Init())

 return false;

 return true;

}

void InitDirect3DApp::OnResize()

{

 D3DApp::OnResize();

}

void InitDirect3DApp::UpdateScene(float dt)

{

}

void InitDirect3DApp::DrawScene()

{

 assert(md3dImmediateContext);

 assert(mSwapChain);

 md3dImmediateContext->ClearRenderTargetView(mRenderTargetView, reinterpret\_cast<const float\*>(&Colors::Blue));

 md3dImmediateContext->ClearDepthStencilView(mDepthStencilView, D3D11\_CLEAR\_DEPTH|D3D11\_CLEAR\_STENCIL, 1.0f, 0);

 HR(mSwapChain->Present(0, 0));

}