# 附录A：Windows编程入门

在使用Direct3D API（Application Programming Interface，应用程序编程接口）时，我们需要创建一个带有主窗口的Windows（Win32）应用程序，在该主窗口上绘制3D场景。本附录主要介绍使用底层Win32 API编写Windows应用程序的入门知识。简单地说，Win32 API就是一组以C语言形式提供给我们的用来创建Windows应用程序的底层函数和结构体的集合。例如，我们可以通过填充一个Win32 API结构体**WNDCLASS**来定义一个窗口类，使用Win32 API函数**CreateWindow**来创建一个窗口，使用Win32 API函数**ShowWindow**让Windows显示某个特定窗口。

Windows编程涉及的内容极广，本附录只能是对Direct3D必需用到的那部分知识做一简单介绍。如果读者希望对Win32 API编程有更深入的了解，那么可以参阅Charles Petzold 编著的《Windows 程序设计（第5版）》，该书是一领域的精典著作。另外，MSDN是一个非常有用的资料库，它包含了大量的微软技术文章。MSDN通常包含在微软的Visual Studio集成开发环境中，不过你也可以通过网址[www.msdn.microsoft.com](http://www.msdn.microsoft.com/)在线阅读。一般来说，当你想要了解一个函数或结构体的用法时，查阅MSDN是最好的选择，因为它里面包含了非常完整的文档。对于本附录中提及但没有详细说明的Win32函数或结构体，读者可以自己查阅MSDN。

## 学习目标：

1. 学习和了解Windows编程所采用的事件驱动编程模型。
2. 学习 Direct3D必用的基本Windows应用程序代码。

注意：为了避免混淆， 我们用大写字母“W”开头的“Windows”表示Windows操作系统，用小写字母“w”开头的“windows”表示运行在Windows操作系统上的窗口程序。

## A.1 概述

顾名思义，Windows编程的主题之一就是编写窗口程序。一个Windows应用程序的许多部分都是窗口，比如主应用程序窗口、菜单、工具栏、滚动条、按钮和其他对话框控件。所以，一个Windows应用程序通常包含多个窗口。下面的几个小节会对Windows编程中的一些基本概念做一简单概述。在我们开始更全面的讨论之前，读者应该先熟悉一些Windows编程中的概念。

### A.1.1 资源

在Windows中，多个应用程序可以并行运行。所以，像CPU周期、内存、甚至显示器这样的硬件资源都必须由多个应用程序共享。为了避免多个应用程序在无序状态下访问和修改资源而造成混乱，Windows禁止应用程序直接访问硬件。Windows的主要任务之一就是管理当前实例化的应用程序，处理多个应用程序之间的资源分配。因此，为了不让我们的应用程序影响其他正在运行的应用程序，我们必须通过Windows来访问事件资源。例如，要显示一个窗口，我们必须调用Win32 API函数**ShowWindow**，而不是直接向显存写入数据。

### A.1.2 事件、消息队列、消息和消息循环

Windows应用程序采用事件驱动模型编程（event-driven programming model）。通常，Windows应用程序会“坐等”事件的发生（**注意**：应用程序可以实现空闲处理（Idle processing）；也就是，在没有任何事件发生执行一些特定的任务）。事件可由多种方式引发；一些常见的事件包括敲击键盘、点击鼠标，以及窗口的创建、缩放、移动、关闭、最小化、最大化或者显示/隐藏。

当一个事件发生时，Windows会向引发事件的应用程序发送一条消息（message），并将消息加入到应用程序的消息队列（message queue）中。消息队列是一个专门用来存储Windows消息的简单的优先队列（priority queue）。应用程序会在一个消息循环中不断地检查消息队列，将收到的消息分发给特定窗口的消息处理函数。（记住，一个应用程序可以包含多个窗口。）每个窗口都有一个与其关联的消息处理函数（每个窗口都有一个消息处理函数（window procedure，直译为窗口过程），而多个窗口可以共享同一个消息处理函数；所以，我们不必为每个窗口都编写一个唯一的消息处理函数。除非不同的窗口要实现不同的功能，处理不同的消息，或者对相同的消息做出不同的反应，我们才有必要为些窗口编写不同的消息处理函数）。消息处理函数是我们自己定义的函数，它包含具体的消息处理代码。例如，我们可能希望在用户按下ESC键时销毁窗口，那么在我们的消息处理函数中就应该包含如下代码：

case WM\_KEYDOWN:

if( wParam == VK\_ESCAPE)

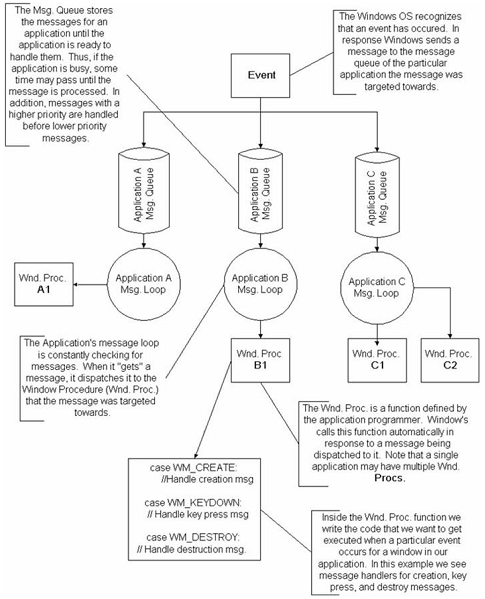
DestroyWindow(ghMainWnd);

return 0;

那些没有被我们的消息处理函数处理的消息都应该被转交给**DefWindowProc**函数来处理，该函数是Win32 API提供的默认消息处理函数。

综上所述，用户操作或应用程序的一些内部行为会引发事件。操作系统会找到引发事件的应用程序，并向应用程序发送一条消息，并把消息加入到应用程序的消息队列中。应用程序不断地检查消息队列。每收到一个消息，应用程序都会将消息分发到与窗口关联的消息处理函数中。最后，消息处理函数执行与当前消息对应的程序指令。

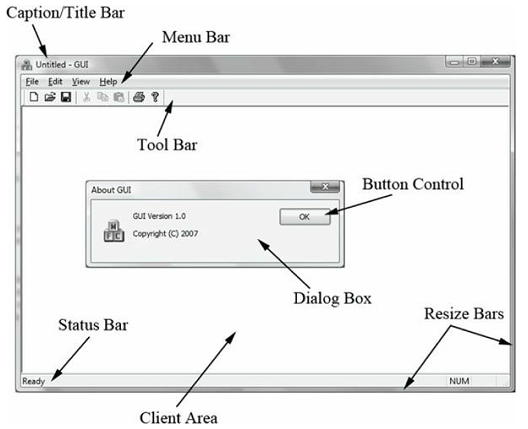
图 A.1 总结了事件驱动模型编程。



**图A.1：事件驱动模型编程。**

### A.1.3 GUI

许多Windows程序都为用户提供了易于操作的GUI（Graphical User Interface，图形化用户界面）。一个典型的Windows应用程序会包含一个主窗口、一个菜单栏、一些工具栏和一些其他的控制窗口。图A.2标出了一些常见的GUI元素。在Direct3D游戏编程中，我们不会用到那些GUI。对于我们来说只有主窗口的客户区有用，我们要在客户区上渲染3D场景。



**图A.2：典型的Windows应用程序GUI。客户区是指应用程序的整个白色矩阵区域，该区域通常用来显示用户数据（比如文本、图像或视频）。当我们编写Direct3D应用程序时，我们要在窗口的客户区中渲染3D场景。**

### A.1.4 Unicode

本质上，Unicode（<http://unicode.org/>）使用16位数表示字符。Unicode字符集很大，它可以支持世界上很多国家的语言文字或其他符号。在C++中，我们用宽字符类型**wchar\_t**来表示Unicode字符。在32和64位Windows中，**wchar\_t**都是16位数。当使用宽字符时，我们必须在字符串文本前面加上一个大写字母“L”。例如：

const wchar\_t\* wcstrPtr = L"Hello, World!";

“L”告诉编译器把一字符串文本看作Unicode字符串（即，以**wchar\_t**代替**char**）。另一个重要问题是我们必须使用字符串函数的Unicode版本。例如，在获取一个字符串长度时，我们必须用**wcslen**代替**strlen**；在复制一个字符串时，我们必须用**wcscpy**代替**strcpy**；在对两个字符串进行比较时，我们必须用**wcscmp**代替**strcmp**。这些函数的Unicode版本使用的都是**wchar\_t**指针，而不是**char**指针。C++标准库也为它的string类提供了一个Unicode版本：**std::wstring**。另外，在Windows头文件中还定义了：

typedef wchar\_t WCHAR; // wc, 16-bit UNICODE character

## A.2 基本的Windows应用程序

下面是一个完全可以运行的Windows程序，代码很简单，读者通过代码中的注释了解它们的含义。我们将在下一节详细讲解些代码。做为一个练习，我们建议读者在你的开发工具中创建一个工程，手工输入些代码，然后编译运行这个程序。注意，如果你使用的是Visual C++，那么在选择工程类型时必须是“Win32 application project”，而不能是“Win32 console application project”。

// 包含Windows头文件；这个文件中包含所有Win32 API声明的

// 结构体、类型和函数，是使用Win32 API必须的基本条件。

#include <windows.h>

// 主窗口句柄；用于标示创建的窗体。

HWND ghMainWnd = 0;

// 封装了初始化一个Windows应用程序的代码。如果初始化成功则返回true，

// 否则则返回false。

bool InitWindowsApp(HINSTANCE instanceHandle,int show);

// 封装了消息循环的代码

int Run();

// 消息处理函数用于处理窗口收到的消息。

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

// 在Windows中，WinMain相当于普通C++编程中的main函数

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

PSTR pCmdLine, int nShowCmd)

{

// 首先调用InitWindowsApp函数创建并初始化主应用程序窗口，

// 该函数的参数是hInstance和nShowCmd。

if(!InitWindowsApp(hInstance, nShowCmd))

return 0;

// 之后进入消息循环，直到接收到WM\_QUIT消息后才退出循环。

return Run();

}

bool InitWindowsApp(HINSTANCE instanceHandle,int show)

{

// 首先填写一个WNDCLASS结构体，描述窗口的基本属性。

WNDCLASS wc;

wc.style = CS\_HREDRAW | CS\_VREDRAW;

wc.lpfnWndProc = WndProc;

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

wc.hInstance = instanceHandle;

wc.hIcon = LoadIcon(0, IDI\_APPLICATION);

wc.hCursor = LoadCursor(0, IDC\_ARROW);

wc.hbrBackground = (HBRUSH)GetStockObject(WHITE\_BRUSH);

wc.lpszMenuName = 0;

wc.lpszClassName = L"BasicWndClass";

// 然后注册这个WNDCLASS示例，这样就

// 可以基于它创建一个窗口。

if(!RegisterClass(&wc))

{

MessageBox(0, L"RegisterClass FAILED",0, 0);

return false;

}

// 注册了WNDCLASS示例后，我们就可以使用CreateWindow方法创建一个窗口。

// 这个方法返回一个指向新创建的窗口的句柄(HWND)。如果创建失败，这个

// 句柄为零。我们使用这个HWND变量引用由Windows维护的主程序窗口。

// 我们必须保存这个窗口句柄，因为许多API函数都要求传入窗口句柄，明确所要操纵的窗口对象。

ghMainWnd = CreateWindow(

L"BasicWndClass", // Registered WNDCLASS instance to use.

L"Win32Basic", // window title

WS\_OVERLAPPEDWINDOW, // style flags

CW\_USEDEFAULT, // x-coordinate

CW\_USEDEFAULT, // y-coordinate

CW\_USEDEFAULT, // width

CW\_USEDEFAULT, // height

0, // parent window

0, // menu handle

instanceHandle, // app instance

0); // extra creationparameters

if(ghMainWnd == 0)

{

MessageBox(0, L"CreateWindow FAILED", 0, 0);

return false;

}

// 最后一步是使用下面的两个函数显示并更新新创建的窗口。

// 将刚刚创建的窗口句柄传递给这两个函数，使Windows知道要显示和更新哪个窗口。

ShowWindow(ghMainWnd, show);

UpdateWindow(ghMainWnd);

return true;

}

int Run()

{

MSG msg = {0};

// 直到接收到WM\_QUIT消息才会退出循环。当收到一个WM\_QUIT消息时，

// GetMessage函数会返回0，并退出循环。当出现错误时，

// GetMessage函数会返回−1。注意,若没有收到消息，

// GetMessage方法会让线程进入休眠状态。

BOOL bRet = 1;

while( (bRet = GetMessage(&msg, 0, 0, 0)) != 0)

{

if(bRet == -1)

{

MessageBox(0, L"GetMessage FAILED",L"Error", MB\_OK);

break;

}

else

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

return (int)msg.wParam;

}

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)

{

// 处理指定的消息。注意，若自己处理消息，就必须返回0。

switch( msg)

{

// 点击鼠标左键，显示一个消息框。

case WM\_LBUTTONDOWN:

MessageBox(0, L"Hello, World", L"Hello", MB\_OK);

return 0;

// 当按下Escape时，会销毁主程序窗口

case WM\_KEYDOWN:

if( wParam == VK\_ESCAPE)

DestroyWindow(ghMainWnd);

return 0;

// 接收到WM\_DESTROY消息后，发送退出信息，终止消息循环。

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

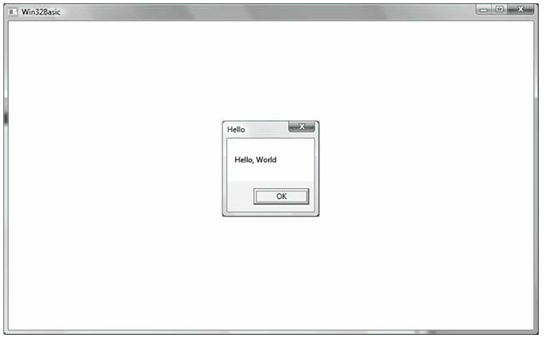
}

// 将前面未处理的其他信息发送到默认的信息处理程序。

// 注意，这个程序返回的是DefWindowProc的返回值。

return DefWindowProc(hWnd, msg, wParam, lParam);

}



**图 A.3：上述程序的屏幕截图。注意，当鼠标单击窗口客户区时会弹出一个消息框。还可以试着按下ESC键退出程序。**

## A.3 程序解析

我们将按照从上到下的顺序分析些代码，依次讲解遇到的每个函数。读者在阅读以下几节的过程中，可以时常参照上面列出的代码。

### A.3.1 头文件、全局变量和函数原型

我们要做的第一件事情是包含windows.h头文件，获得windows.h头文件中声明的结构体、类型和函数，是使用Win32 API必须的基本条件。

#include <windows.h>

第二条语句定义了一个**HWND**类型的全局变量，该类型用于表示“窗口句柄”。在Windows编程中，我们经常使用句柄来引用那些由Windows内部维护的对象。在本例中，我们使用这个**HWND**变量引用由Windows维护的主程序窗口。我们必须保存这个窗口句柄，因为许多API函数都要求传入窗口句柄，明确所要操纵的窗口对象。例如，在调用**UpdateWindow**函数时需要传入一个**HWND**参数，指定所要刷新的窗口。如果我们不传入窗口句柄，那该函数就无法确定要刷新哪个窗口。

HWND ghMainWnd = 0;

下面的3行是函数声明。简单地说，**InitWindowsApp**负责创建和初始化主程序窗口， **Run**封装了程序的消息循环，**WndProc**是主窗口的消息处理函数。在随后的几个小节中，当我们调用些函数时，会对它们进行更为详细的分析和说明。

bool InitWindowsApp(HINSTANCE instanceHandle,int show);

int Run();

LRESULT CALLBACK WndProc(HWND hWnd, UINT msg,WPARAM wParam, LPARAM lParam);

### A.3.2 WinMain

Windows中的**WinMain**函数与普通C++编程中的**main**函数的作用相同。**WinMain**函数的原型如下：

int WINAPI

WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

PSTR pCmdLine, int nShowCmd)

1．**hInstance**：当前应用程序的实例句柄。它是标识和引用当前应用程序的一种方式。记住，同一个Windows应用程序可能会并行运行多个实例（比如同时运行多个Microsoft Word实例），所以在引用某个特定实例时它非常有用。

2．**hPrevInstance**：该参数的值为0。它是16位Windows遗留下来的一个参数，Win32编程已不再使用。

3．**pCmdLine**：程序启动时传入的命令行参数字符串。

4．**nShowCmd**：指定应用程序窗口的显示方式。一些常用的值有：**SW\_SHOW**（以当前的尺寸和位置显示窗口）、**SW\_SHOWMAXIMIZED**（最大化）和**SW\_SHOWMINIMIZED**（最小化）。要了解该参数的所有选项值，请参阅MSDN。

如果**WinMain**函数调用成功，那么它将返回**WM\_QUIT**消息的**wParam**成员。如果函数在未进入消息循环就已退出，那么它将返回0。标识符**WINAPI**的定义如下：

#define WINAPI \_\_stdcall

它指定了函数的调用方式，即以何种方式处理堆栈上的函数参数。

### A.3.3 WNDCLASS与注册

在**WinMain**中我们调用了函数**InitWindowsApp**。也许你能猜到，该函数用于完成程序的初始化工作。下面让我们进一步分析一下该函数的实现过程。**InitWindowsApp**返回一个bool值——当初始化成功时返回**true**，否则返回**false**。在**WinMain**函数中，我们为**InitWindowsApp**传入了应用程序的实例句柄**hInstance**和窗口的显示方式**nShowCmd**，这两个参数可以从**WinMain**的参数列表中得到。

if(!InitWindowsApp(hInstance, nShowCmd))

要初始化一个窗口，第一步是要填写一个**WNDCLASS**（window class，窗口类）结构体，描述窗口的基本属性。该结构体的定义如下：

typedef struct \_WNDCLASS{

UINT style;

WNDPROC lpfnWndProc;

int cbClsExtra;

int cbWndExtra;

HANDLE hInstance;

HICON hIcon;

HCURSOR hCursor;

HBRUSH hbrBackground;

LPCTSTR lpszMenuName;

LPCTSTR lpszClassName;

} WNDCLASS;

1．**style**：指定窗口类的样式。在本例中，我们使用了**CS\_HREDRAW**和**CS\_VREDRAW**的组合。这两个二进制位（bit）标志值表明当窗口的宽度或高度发生变化时，窗口将被重绘。要了解该参数的所有选项值，请参阅MSDN。

wc.style = CS\_HREDRAW | CS\_VREDRAW;

2．**lpfnWndProc**：与**WNDCLASS**关联的消息处理函数指针。窗口使用的消息处理函数完全由**WNDCLASS**中的这个参数决定。也就是说，如果我们使用同一个**WNDCLASS**来创建两个窗口，那么这两个窗口将使用同一个消息处理函数。反之，如果我们希望两个窗口使用不同的消息处理函数，那我们就必须定义两个不同的**WNDCLASS**实例，分别指定不同的消息处理函数。有关消息处理函数的内容请参见A.3.6 节。

wc.lpfnWndProc = WndProc;

3．**cbClsExtra**与**cbWndExtra**：用于存储附加数据。这两个参数通常不会用到，所以都设为0。

wc.cbClsExtra = 0;

wc.cbWndExtra = 0;

4．**hInstance**：应用程序实例句柄。回顾前文，应用程序实例句柄是操作系统传给**WinMain**函数的第1个参数。

wc.hInstance = instanceHandle;

5．**hIcon**：用于指定显示在窗口标题栏上的图标。你可以自己设计图标，也可以使用Windows内置的几个图标；具体内容请参见MSDN。这里使用默认的应用程序图标：

wc.hIcon = LoadIcon(0, IDI\_APPLICATION);

6．**hCursor**：用于指定鼠标位于窗口客户区时显示的光标。与**hIcon**类似，你可以自己设计光标，也可以使用Windows 内置的几个光标；具体内容请参见MSDN。这里使用标准的“箭头”光标：

wc.hCursor = LoadCursor(0, IDC\_ARROW);

7．**hbrBackground**：用于指定窗口客户区的背景色。在本例中，我们调用Win32函数GetStockObject返回一个内置的白色画刷句柄；如果你想了解其他的内置画刷，请参见 MSDN。

wc.hbrBackground = (HBRUSH)GetStockObject(WHITE\_BRUSH);

8．**lpszMenuName**：指定窗口菜单。由于我们的应用程序没有菜单， 所以将该参数设为 0。

wc.lpszMenuName = 0;

9．**lpszClassName**：指定窗口类的名称（也就是，为**WNDCLASS**实例起一个名字），任何有效的字符串都可以。在本例中，我们将它命名为“BasicWndClass”。在随后的代码中，我们将通过个名称来引用相应的**WNDCLASS**实例。

wc.lpszClassName = L"BasicWndClass";

随后，我们要调用**RegisterClass**函数把填好的**WNDCLASS**实例注册到Windows中。该函数接收一个指向**WNDCLASS**结构体的指针。在调用失败时返回0。

if(!RegisterClass(&wc))

{

MessageBox(0, L"RegisterClass FAILED", 0, 0);

return false;

}

### A.3.4 创建和显示窗口

当我们把**WNDCLASS**实例注册到Windows系统之后，就可以依据该窗口类的描述来创建实际的窗口对象了。另外，我们可以通过窗口类的名字（**lpszClassName**）来引用注册后的WNDCLASS 实例。用于创建窗口对象的函数为**CreateWindow**，它的函数原型如下：

HWND CreateWindow(

LPCTSTR lpClassName,

LPCTSTR lpWindowName,

DWORD dwStyle,

int x,

int y,

int nWidth,

int nHeight,

HWND hWndParent,

HMENU hMenu,

HANDLE hInstance,

LPVOID lpParam

);

1．**lpClassName**：已注册的**WNDCLASS**实例的名字，该实例描述了所要创建的窗口对象的部分特征。

2．**lpWindowName**：窗口的名字；即显示在窗口标题栏上的字符串名称。

3．**dwStyle**：定义窗口的样式。**WS\_OVERLAPPEDWINDOW**是以下几个标志值的组合：**WS\_OVERLAPPED**、**WS\_CAPTION**、**WS\_SYSMENU**、**WS\_THICKFRAME**、**WS\_MINIMIZEBOX**和**WS\_MAXIMIZEBOX**。这些标志值描述了所要创建的窗口的特征。 详情请参见MSDN。

4．**x**：窗口左上角在屏幕上的x坐标。可以将该参数设为CW\_USEDEFAULT，Windows会为窗口选择一个适当的默认坐标值。

5．**y**：窗口左上角在屏幕上的y坐标。可以将该参数设为CW\_USEDEFAULT，Windows会为窗口选择一个适当的默认坐标值。

6．**nWidth**：窗口宽度，单位为像素。可以将该参数设为CW\_USEDEFAULT，Windows会为窗口选择一个适当的默认宽度。

7．**nHeight**：窗口高度，单位为像素。可以将该参数设为CW\_USEDEFAULT，Windows会为窗口选择一个适当的默认高度。

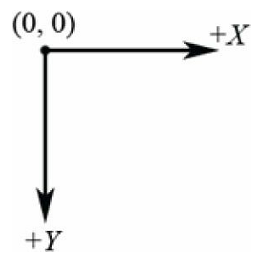
8．**hWndParent**： 当前窗口的父窗口句柄。由于本例中的窗口没有任何父属窗口，所以该参数设为0。

9．**hMenu**：菜单句柄。由于本例中的窗口没有菜单，所以该参数设为0。

10．**hInstance**：与窗口关联的应用程序实例句柄。

11．**lpParam**：传递给**WM\_CREATE**消息处理函数的用户自定义数据的指针。当创建窗口时，操作系统会向窗口发送一个**WM\_CREATE**消息。该消息会在**CreateWindow**方法返回前发出。如果希望在创建窗口执行某些操作（比如，完成某些对象的初始化工作），那么就需要处理窗口的**WM\_CREATE**消息。

**注意**：当我们为窗口指定x、y坐标时，这些坐标位置是相对于屏幕左上角的。而且，x轴的正方向水平向右，y轴的正方向垂直向下。图A.4展示的坐标系称为屏幕坐标系（screen coordinate system），或屏幕空间（screen space）。



**图 A.4：屏幕空间。**

**CreateWindow**函数将返回创建后的窗口句柄（一个**HWND**变量）。如果创建失败，则该句柄的值为0（空句柄）。记住，句柄是引用窗口的一种方式，而窗口由Windows来管理。许多API函数都要求传入窗口句柄，以明确所要操纵的窗口对象。

ghMainWnd = CreateWindow(L"BasicWndClass", L"Win32Basic",

WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT,

CW\_USEDEFAULT, CW\_USEDEFAULT,

0, 0, instanceHandle, 0);

if(ghMainWnd == 0)

{

MessageBox(0, L"CreateWindow FAILED", 0, 0);

return false;

}

在InitWindowsApp中调用的最后两个函数用于控制窗口的显示。首先我们调用**ShowWindow**，并且将刚刚创建的窗口句柄传递给它，使Windows知道要显示哪个窗口。我们传入的另一个整数值表示窗口的初始显示状态（例如，最小化、最大化等等）。该值应设为**nShowCmd**（它是**WinMain**函数的第4个参数）。当初次显示窗口时，必须调用UpdateWindow方法，对窗口进行一次手动刷新。该函数只接收一个参数，指定所要刷新的窗口的句柄。

ShowWindow(ghMainWnd, show);

UpdateWindow(ghMainWnd);

到此为止，我们在**InitWindowsApp**函数中的初始化工作就已经完成了；如果一切顺利的话，该函数将返回**true**。

### A.3.5 消息循环

在成功完成初始化工作之后，我们开始编写程序的核心部分——消息循环。在本例中， 我们将消息循环封装在了**Run**函数中。

int Run()

{

MSG msg = {0};

BOOL bRet = 1;

while( (bRet = GetMessage(&msg, 0, 0, 0)) != 0)

{

if(bRet == -1)

{

MessageBox(0, L"GetMessage FAILED",L"Error", MB\_OK);

break;

}

else

{

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

return (int)msg.wParam;

}

在**Run**函数中做的第一件事情是声明一个**MSG**类型的变量**msg**。**MSG**是一个用于表示Windows 消息的结构体，它的原型如下：

typedef struct tagMSG {

HWND hwnd;

UINT message;

WPARAM wParam;

LPARAM lParam;

DWORD time;

POINT pt;

} MSG;

1．**hwnd**：接收消息的窗口的句柄。

2．**message**：用于标识特定消息的预定义常量值（例如，WM\_QUIT）。

3．**wParam**：与消息相关的附加信息，参数的取值依具体消息而定。

4．**lParam**：与消息相关的附加信息，参数的取值依具体消息而定。

5．**time**：消息被送入消息队列时的时间。

6．**pt**：当消息被送入消息队列时，鼠标在屏幕空间中的x、y坐标。

随后，我们进入消息循环。**GetMessage**函数从消息队列中检索消息，使用消息的具体内容来填充msg参数。**GetMessage**函数的第2、3、4个参数均设为0。当出现错误时，**GetMessage**函数会返回−1。当收到一个**WM\_QUIT**消息时，**GetMessage**函数会返回0，此时可以终止消息循环的运行。当**GetMessage**返回其他值时，需要调用另外两个函数：**TranslateMessage**和**DispatchMessage**。TranslateMessage用于进行某些键盘消息的转换；确切地说是将虚拟键盘码转换为字符信息。最后，DispatchMessage将消息分发给特定的消息处理函数。

如果应用程序以**WM\_QUIT**消息正常退出，那么**WinMain**函数应以**WM\_QUIT**消息的**wParam**参数作为最终的返回值（退出码）。

### A.3.6 消息处理函数

我们前面提到，消息处理函数（window procedure，直译为窗口过程）用于对窗口收到的消息做出响应，执行与当前消息对应的程序指令。在本例中，我们的消息处理函数为**WndProc**，其原型如下：

LRESULT CALLBACK

WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);

该函数返回一个**LRESULT**（实际上是一个整数）值，表明函数的调用结果是成功还是失败。**CALLBACK**标识符说明该函数是一个回调函数（callback function），Windows将在应用程序的代码空间之外调用个函数。你可以从本例的源代码中看到，我们从未直接调用过个消息处理函数——当窗口需要处理一个消息时，Windows会替我们调用个函数。

在消息处理函数的签名（signature）中包含4个参数：

1．**hWnd**：接收消息的窗口的句柄。

2．**msg**：用于标识特定消息的预定义常量值。例如，退出消息为**WM\_QUIT**。前缀 WM 表示“window message （窗口消息）”。预定义的窗口消息有一百多个，具体请参见MSDN。

3．**wParam**：与消息相关的附加信息，参数的取值依具体消息而定。

4．**lParam**：与消息相关的附加信息，参数的取值依具体消息而定。

我们的消息处理函数处理了3个消息：**WM\_LBUTTONDOWN**、**WM\_KEYDOWN**和**WM\_DESTROY**。**WM\_LBUTTONDOWN**是用户在窗口客户区单击鼠标左键时发出的消息。**WM\_KEYDOWN**是用户按下某个按键时发出的消息。**WM\_DESTROY**是窗口被销毁时发出的消息。

我们的代码相当简单；当窗口收到**WM\_LBUTTONDOWN**消息时会弹出一个消息框，显示“Hello, World”字符串：

case WM\_LBUTTONDOWN:

MessageBox(0, L"Hello, World", L"Hello", MB\_OK);

return 0;

当窗口收到**WM\_KEYDOWN**消息时，测试用户按下的是否为ESC键。如果是ESC键，我们就调用**DestroyWindow**函数销毁主程序窗口。此时传给消息处理函数的**wParam** 参数包含了用户所按按键的虚拟键盘码。每个物理按键都有一个或多个与之对应的虚拟键盘码，它可以被视为物理按键在C++程序中的标识符。windows.h头文件包含了所有的虚拟键盘码常量，我们可以使用些常量来测试某一按键是否被按下；例如，我们可以使用虚拟键盘码常量**VK\_ESCAPE**来测试ESC键是否被按下：

case WM\_KEYDOWN:

if( wParam == VK\_ESCAPE)

DestroyWindow(ghMainWnd);

return 0;

记住，**wParam**和**lParam**参数用于指定消息的附加信息。对于WM\_KEYDOWN消息来说，wParam参数包含了与按键对应的虚拟键盘码。MSDN详述了每个Windows消息的**wParam**和**lParam**参数含义。

当窗口被销毁时，我们需要使用PostQuitMessage函数发送一个**WM\_QUIT**消息（它将终止消息循环的运行）：

case WM\_DESTROY:

PostQuitMessage(0);

return 0;

在我们的消息处理函数的最后，调用了**DefWindowProc**函数。该函数是默认的消息处理函数。在本例中，我们只处理了3个消息；其他所有的消息都按DefWindowProc函数中定义的默认行为来处理，我们不必亲自处理所有的消息。例如，本例中的最小化、最大化、窗口缩放和关闭功能都是由默认的消息处理函数来完成的，我们没有亲自处理些消息。

### A.3.7 MessageBox函数

我们还有最后一个API函数没有讲解，也就是**MessageBox**函数。该函数可以非常方便地向用户显示提示信息，并获得一些简单的输入。**MessageBox**函数的原型如下：

int MessageBox(

HWND hWnd, // Handle of owner window, may specify null.

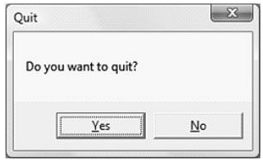
LPCTSTR lpText, // Text to put in the message box.

LPCTSTR lpCaption, // Text for the title of the message box.

UINT uType // Style of the message box.

);

**MessageBox**函数的返回值取决于消息框的类型。要了解所有的返回值和消息框类型，请参见MSDN；图A.5展示了一种带有“Yes/No”按钮的消息框。



**图A.5：带有“Yes/No”按钮的消息框。**

## A.4 改进后的消息循环

游戏与传统的Windows应用程序之间有很大的区别。通常，游戏会主动重绘界面，不停地更新窗口，而不是坐等消息的到来。传统的消息循环会造成一个问题，当消息队列没有消息时，**GetMessage**函数会让线程进入休眠状态，等待消息到来。而在游戏中，我们不希望这种情况发生；当没有Windows 消息需要处理时，我们希望游戏运行自身的代码（比如，渲染3D场景、处理AI等等）。所以，我们要对消息循环做一些修改，用**PeekMessage**函数代替原先的**GetMessage**函数。**PeekMessage**函数会在没有消息时直接返回，不阻塞线程。我们改进后的消息循环如下：

int Run()

{

MSG msg = {0};

while(msg.message != WM\_QUIT)

{

// If there are Window messages then process them.

if(PeekMessage( &msg, 0, 0, 0, PM\_REMOVE ))

{

TranslateMessage( &msg );

DispatchMessage( &msg );

}

// Otherwise, do animation/game stuff.

else

{

}

}

return (int)msg.wParam;

}

在声明**msg**变量之后，我们会进入了一个无限循环。我们首先调用API函数**PeekMessage**检查消息队列中是否存在消息。有关该函数的参数描述请参见 MSDN。如果消息队列中存在消息，则**PeekMessage**返回**true**，我们执行与先前同样的消息处理；反之，如果没有消息，则**PeekMessage**返回false，我们运行游戏自身的代码。

## A.5 小结

1．在使用 Direct3D时，我们必须创建一个带有主窗口的Windows应用程序，在主窗口上渲染3D场景。而且，要为游戏创建一个特殊的消息循环。在有消息处理消息；在没有消息执行游戏自身的代码逻辑。

2．多个Windows应用程序可以并行运行，所以Windows必须管理那些由多个应用程序共享的资源，对应用程序产生的消息进行疏导。当一个事件（按键、鼠标单击、计时器等等）产生时，消息即被发送到应用程序的消息队列中。

3．每个Windows应用程序都有一个消息队列，用来存储应用程序收到的消息。应用程序的消息循环会不断地检查消息队列，当有消息出现时，它会把消息分发给相应的消息处理函数。注意，一个应用程序可以包含多个窗口。

4．消息处理函数是一个由我们自己编写的特殊的回调函数，当窗口收到消息时，Windows会自动调用与窗口对应的消息处理函数。在消息处理函数中，我们为特定的消息编写代码，使窗口在收到特定的消息执行相应的程序逻辑。对于那些未在消息处理函数中处理的消息，应转交给默认的消息处理函数来执行默认的行为。