Windows 游戏编程大师技巧

电子版 version 0.1

2D 和 3D 游戏编程基础

ANDRE LAMOTHE 著
曲文卿 姚君山 钟湄莹等译

kofight 制作

录入者名单:

本书目前大部分内容来自http://user.7host.com/ecapital/易都网

第一部分 Windows编程基础

第一章 无尽之旅

Windows 编程是一场由来已久并还在进行着的战争。开始时，游戏程序拒绝 Windows 平台，但正如 Borg 所言：“反对无效.....”，我也赞同这一观点。本章就 Windows 的发展进行一下回顾。

- 游戏的历史
- 游戏类型
- 游戏编程的基本要素
- 使用工具
- 一个游戏的例子：FreakOut

历史一瞥

在 60 年代的某个时候，第一台台式计算机问世，当时运行在 Unix 机器上的 Core Wars，是最早的计算机游戏之一。当 70 年代的黄金岁月到来时，在全世界的台式计算机和小型计算机上流行着文本游戏和粗糙的图片游戏。有趣的是从那以后，大多数游戏开始网络化。我是说，90%的游戏程序 MUD(Multi-User Dungeons)或类似的模拟游戏，如 Star Trek 和战争模拟游戏。但是，直到精彩的 Pong 游戏问世，大众才真正品尝到计算机游戏的乐趣。由 Nolan Busnell 设计的 Atari 计算机的问世，使单人的视觉游戏一夜之间充斥市场。

而后，在 1976～1978 年间，TRS-80、Apple、Atari 800 等型计算机开始冲击市场，这些是消费者买得起的第一代计算机。当然，读者也可以买类似 Altair 8000 型的组装机，但是又有谁乐意进行它们的组装呢？无论如何，这些计算机有各自的优缺点，Atari 800 是当时功能最强大的计算机(我也保证，我可以写一个 Wolfenstein 程序在它上面运行)。TRS-80 最商业化，而 Apple 机的市场最好。

慢慢地，游戏开始冲击计算机相关产业方面的市场，一夜之间，产生了许多年轻的百万富翁。制作者只需要一个类似“月亮领地”或 Pong 类型的好游戏，就可以突然致富！从那时，游戏开始像真正的计算机游戏，而且只有百十人知道如何编写游戏，当时绝对没有这类的指导书，只是有人不时半地下地出版一些 50～100 页的有许多令人费解之处的小册子，另外也曾在 Byte 杂志上有过文章，但是，大多时候，读者必须靠自己。

80 年代是游戏升温的年代。第一代 16 位计算机问世，如：IBM PC 及其兼容机、Mac、Atari ST、Amiga 500 等等。这是游戏变得好看的时候，甚至有些 3D 游戏如 Wing Commander、Fight Simulator 等开始出现在市场上。但是，PC 机肯定仍然落后于游戏机。在 1985 年以前，Amiga 500 和 Atair ST 作为最终游戏机占有绝对的支配地位。但是 PC 机由于低廉的价格和其商业用途开始大众化，并且最终必须将是具有最大市场基础的计算机——不考虑它的技术和质量——一统江湖。

在 90 年代初期，IMB PC 及其兼容机是主流。随着微软的 Windows 3.0 的发行，Apple Macintosh 寿终正寝。PC 是“工作者的计算机”。读者可以真正在上面进行游戏、编程、开机并连接其他东西。我想这就是为什么会有这么多人沉醉于 PC 而不是 Mac 的原因。一句话，读者不可能从 Mac 中寻求乐趣。

但是 PC 在图像或声音上还依然落后，PC 机看起来像没有足够的马力使游戏如同在 Amiga 或者游戏控制台上表现得那样好。

而后曙光终于来临.....

在 1993 年后期，Id Softwere 发行了 DOOM 作为 Wolfenstein 3D(是由 Id 编写的第一批 3D 游戏之一)的换代产品。PC 机开始成为家用微机市场玩游戏或编程的选择，直到现在也是。DOOM 表明，只要读者有足够的才智，就可以在微机上面作任何事情。这点非常重要，记住，没有任何东西可以替代想像力和决心，只要读者认为可能，它就能实现。

在 DOOM 的疯狂冲击下，Microsoft 开始重新估计它在游戏和游戏编程上的地位，它意识到娱乐产业的巨大，并且只会更大。它开始制定巨大的计划涉足游戏，以使它得以在这个产业中分一杯羹。

问题是即使 Windows 95 的实时视听能力也表现得很恶劣，于是微软发行了 Win-G 以解决视觉方面的问题，Win-G 被宣布为游戏编程和图形子系统的最终解决方案，而实际上它不过只是用位图画的一些图像而已。一年之后 Microsoft 却又否认它的存在。——这可实有其事！

但是，新的图形、声音、输入、网络、3D 系统的工作已经开始，DirectX 诞生了。像以往一样，Microsoft 发行人员宣称它将解决世界上 PC 机平台上所有游戏编程的问题，并且在 Windows 上运行的游戏将同 DOS32 上的游戏一样快，甚至更快。但事实上并非如此。

DirectX 开始的两个版本在实际应用中是失败的，但这并非指技术而言，Microsoft 低估了视觉游戏编程的复杂性及视觉游戏程序员能力。但是到了 DirectX3.0，DirectX 就开始工作得比 DOS 出色！但是那时(1996～1997)许多公司仍然采用 DOS32 平台进行游戏编程，直到 DirectX 版本 5.0 发行，人们才转而使用 DirectX。

现在 DirectX 已经升级到 8.0 版本(本书包含 7.0 版本)，它是一类极强大的 API。的确，读者应当改变一下想法——用 COM(Component Object Model)在 Win32 上编程，不再对整个计算机进行全部控制——但生活就是如此。

利用 DirectX 技术，读者可以创建一个 4GB 地址（或更多）、内存线性连续的、仿 DOS 的虚拟机，读者可以像在 DOS 环境（如果读者喜欢的话）下那样来编程。更为重要的是，现在读者可以即时操纵图像和声音技术的每一个新的片段。这都归功于 DirectX 远见卓识的设计和技术。总之，关于 DirectX 的话题已经足够多了，不久读者就会接触到整个处理所带来的效果。

首先出现的是 DOOM 游戏，它仅用到软件光栅技术，如图 1.1 所示。看一看 Rex Blade 游戏的屏幕表现，它是 DOOM 游戏的一个克隆版本。下一代的 3D 游戏，如 QuakeⅠ、QuakeⅡ和 Unreal 就有了巨大的飞跃。再看一看图 1.2 中 Unreal 游戏的屏幕表现。这个游戏及其类似的游戏，其表现简直令人难以置信。所有这类游戏都采用软件光栅和硬件加速代码来获得最好的游戏表现。提请读者注意的是，Unreal 和 QuakeⅡ游戏至少要运行在配置 VoodooⅡ加速卡的 PentiunⅡ400MHz 机器上才能够达到这么眩目的效果。那么这将把我们带向何方？技术的发展越来越先进，以至空间已成为制约因素。然而，“奇迹”总会涌现。即使诸如 Quake 和 Unreal 这类的游戏需要花费数年才能制作完成，我们仍然相信读者也能够创作出有如此吸引力的游戏。

历史的回顾到此为止，下面让我们转到游戏软件设计的核心上来。

设计游戏

编写视频游戏最难的一个工作是设计。的确，3D 运算很难，但是策划一个有趣的游戏并且进行设计可谓同样困难、重要。谁会关心游戏中是否采用最新的光子跟踪技术呢？

现在，构思一个游戏并不难。难的是它的细节体现、最终实现和相似物体在视觉形象上的差别表现。因此下面概述一下一些基本概念和本人在游戏设计工作中的一些心得、经验。

游戏类型

现在，游戏类型多如政治许诺（只说不做），但可以将它们归入以下几个类型：

类似 DOOM 的单人游戏——这些游戏大部分是全 3D 游戏，你将以角色的身份置身游戏中。DOOM、Hexen、Quake、Unreal、Duke Nukem 3D 以及 Dark Forces 都属于这一类型的游戏。

从技术上讲，它们或许是最难开发的游戏 ，这类游戏要求采用边缘切换技术。

运行游戏——运行游戏可以是 2D 的，也可以是 3D 的，但是近来 3D 的运行游戏越来越多了。运动游戏可以一个人玩。也可以多人玩。运动游戏的图像已经有了很大的改进。也许运动游戏不像单人游戏给人印象深刻，但是它们也很吸引人。当然运行游戏中的人工智能水平却是所有游戏类别中最先进的。

格斗游戏——

街道/枪战/平面游戏——

机械模拟游戏——

生态模拟游戏——

攻略或战争游戏——

交互式故事——

重新流行的游戏——

纯智力和棋牌游戏——

集思广益

一旦读者决定了制作哪一种游戏（这是件简单的事，因为我们知道自己喜欢什么），就到了构思这个游戏的时候了。这完全由读者自己决定，策划一个好游戏并没有一成不变的思维定式。

首先，必须想出一个读者喜欢制作并将它开发成为听上去很酷的、可以实现的，并且其他人能够喜欢的游戏。

当然，读者可以将其他游戏作为模板或出发点来得到启发。不要简单地复制其他产品，但是大致模仿已成功的产品也是可以的。并且，大量阅读科幻书籍和游戏杂志，观察正在卖什么观看大量的电影将有助于产生很酷的故事想法、游戏想法或视频动作。

我通常所做的是和朋友一起坐坐（或者是自己），抛出各种想法，直到出现听上去很酷的想法。然后，开发这个想法，直到似是而非，或者土崩瓦解为止。这令人非常心灰意冷。读者可能会对所有的想法厌烦，然后在两三个小时后放弃。不要灰心，这是件好事，如果一个游戏想法能存活一夜，第二天想起来还喜欢它的话，也许运气就来了。

警告：在这里我想给读者一个告诫：不要贪多嚼不烂！我已经收到上千封游戏编程新手的电子邮件。这些读者一心想在很短的时间内开发出诸如 DOOM 或 Quake 这类高水平的游戏来作为他们的处女作，显然这是不可能的。如果读者在 3～6 个月内能够完成一个 Asteroids 的复制品，就是很幸运的了，因此不要狂热。设定一个可行的目标，尝试考虑一些自己能做的事，因为最后可能只有自己在继续工作，别人都离你而去了。另外让处女作游戏的想法简单些。

下面我们进行一些细节工作。

设计文档和情节图板

一旦读者有了一个游戏想法，就应当将它落实到纸上。现在，当我进行一个大游戏产品时，需要自己编写一个实际的设计文档，但是对于少数游戏来讲，应当做几页的细节。首先一个设计文档是一个游戏的路标或框架。应当编写能想到的、尽可能多的游戏、游戏等级和游戏规则的细节。这样能知道正在做什么以及能按计划做下去。相反，如果想法一直在变化，那么所做的游戏不连贯。

通常，我喜欢将一个简单的故事写下，开始的一或两页可能描写这个游戏是什么游戏，谁是主角、游戏思路是什么、最后游戏如何攻关。然后我决定游戏的核心细节——游戏等级和规则，列出尽可能多的细节。完成后，就一直进行添加或删除，但至少我有一个工作计划。如果想出了 100 条很酷的新思路，我能够一直补充它们，而不会忘记。

很明显，细节的数量由读者决定，但是要写下一些东西，至少是一个游戏梗概。例如，可能读者不会连续地考虑一个完整的设计文档，而是一些大致的游戏等级和规则的框图。图 1.3 是为一个游戏编写的情节图板的例子。没有复杂的细节，只有方便观察和工作的草图。

使游戏具有趣味性

游戏设计的最后部分是实际校验。读者确信游戏具有趣味性并且人们喜欢它吗？还是在自欺欺人？这是一个严重的问题。大约有 10000 个游戏被搁置，9900 个公司歇业，因此要仔细考虑。如果读者完全为之着迷并不顾一切的想玩这个游戏的话，那么已经大功告成。但是如果读者作为一个设计者对该想法冷淡的话，想像一下如何令其他人来对它感兴趣呢！

这里的关键是进行大量的游戏策划和 beta 测试，增加各种类型的非常酷的特性，因为这才是令游戏生动有趣的关键所在。这就如同橡木家具上的精美的手工艺品。人们喜欢这些游戏细节。

游戏的构成

现在来看一下一个视频游戏程序和其他各种程序的区别。视频游戏是一种极其复杂的软件，毫无疑问它们也是最难编写的程序。显然，编写 MS Word 程序要比 Asteroids 游戏难一点，但是编写 Unreal 游戏则要比我所能想像得到的其他任何程序都要难。

这就表示读者应当学习一种新的编程方式，这种方式更有益于实时应用和模拟，而不是读者经常使用的单行的、事件驱动的或顺序逻辑的程序，一个视频游戏基本上是一个连续的循环，它完成逻辑动作，并在屏幕上产生一个图像，通常

是每秒钟 30 幅图或更多，这和电影的放映非常相似。只是读者要按自己的思路创建这个电影。

下面让我们从观察如图 1.4 所示的简化游戏循环开始，下面对图中每个部分作些说明。

第一步：初始化

在这一步中，游戏程序运行的初始化操作和其他程序一样，如内存单元配置、资源采集、从磁盘装载数据等等。

第二步：进行游戏循环

在这一步中，代码运行进入游戏主循环，此时各种游戏动作和情节开始运行，直至用户退出游戏主循环。

第三步：获得玩家的输入信息

在这一步中，处理游戏玩家的输入信息并将其储存到缓存以备下一步人工智能和游戏逻辑使用。

第四步：执行人工智能和游戏逻辑

这部分包括游戏代码的主体部分，执行人工智能、物理系统和常规的游戏逻辑，其结果用于产生下一帧屏幕图像。

第五步：渲染下一帧图像

本步中，玩家输入和游戏人工智能和逻辑执行的结果，用来产生游戏的下一帧动画。这个图像通常放在后备缓存区内，因此无法看到它被渲染的过程。随后该图像被迅速拷贝到显示区中。

第六步：同步显示

许多计算机会因为游戏复杂程度的不同，游戏的速度会加快或减慢。例如，如果屏幕上有 1000 个对象在运行，CPU 的负载就比只有 10 个对象时重得多，因而游戏画面刷新速度也会有所改变，这是不允许的。因此必须确保游戏和最大帧速同步并使用定时器和/或等待函数来维持同步。一般认为 30 帧/秒是最佳的帧速。

第七步：循环

这一步非常简单，只需返回到游戏循环的入口并重新执行上述全部步骤。

第八步：关闭

这一步结束游戏，表示用户结束主体操作或游戏循环，返回操作系统。然而，在用户进行结束之前，用户必须释放所有的资源并刷新系统，这些操作和其他软件所进行的相应操作相同。

读者可能对上述游戏操作的细节感到疑惑。诚然，上面进行解释有点过于简单化，但是它突出了如何进行游戏编程的重点。在大多数情况下，游戏循环是一个包括了大量状态的FSM(Finite State Machine，有限态计算机）。清单1.1是一个相当复杂的C/C++游戏循环实际代码。

程序清单1.1 一个简单的游戏事件循环

```
//defines for game loop states
#define GAME_INIT //the game is initializing
#define GAME_MENU //the game is in the menu mode
#define GAME_STARTING //the game is about to run
#define GAME_RUN //the game is now running
#define GAME_RESTART //the game is going to restart
#define GAME_EXIT //the game is exiting

//game globals
int game_state = GAME_INIT;//start off in this state
int error =0;//used to send errors back to OS

//main begins here

void main()
{
  //implementation of main game loop
  while(game_state != GAME_EXIT)
  {
    //what state is game loop in
    switch(game_state)
    {
      case GAME_INIT: //the game is initializing
      {
        //allocate all memory and resources
        Init();
        //move to menu state
        game_state =GAME_MENU;
      }break;

      case GAME_MENU://the game is in the menu mode
      {
        //call the main menu function and let it switch states
        game_state =Menu();
        //note:we could force a RUN state here
      }break;
```

```
case GAME_STARTING://the game is about to RUN
{
  //this state is optional,but usually used to
  //set things up right before the game is run
  //you might do a little more housekeeping here
  Setup_For_Run();
  //switch to run state
  game_state = GAME_RUN;
}break;

case GAME_RUN: //the game is now running
{
  //this section contains the entire game logic loop
  //clear the display
  Clear();
  //get the input
  Get_Input();
  //perform logic and ai
  Do_Logic();
  //display the next frame of animation
  Render_Frame();
  //synchronize the display
  Wait();
  //the only way that state can be changed is
  //thru user interaction in the
  //input section or by maybe losing the game.
}break;

case GAME_RESTART: //the game is restarting
{
  //this section is a cleanup state used to
  //fix up any loose ends before
  //running again
  Fixup();
  //switch states back to the menu
  game_state = GAME_MENU;
}break;

case GAME_EXIT: //the game is exiting
{
  //if the game is in this state then
  //it's time to bail,kill everything
  //and cross your fingers
  Release_And_Cleanup();
  //set the error word to whatever
```

```
            error = 0;
            //note: we don't have to switch states
            //since we are already in this state
            //on the next loop iteration the code
            //will fall out of the main while and
            //exit back to the OS
          }break;

          default: break;

        }//end switch
      }//end while
      //return error code to operation system
      return(error);
    }//end main
```

尽管清单 1.1 是一个没有任何功能的程序，但通过学习其游戏循环的结构可以获得很好的思路。所有的游戏循环大都按照这个结构的一种形式或另一种形式进行设计。图 1.5 表示了游戏循环逻辑的状态转换图。如读者所见，状态转换是非常连贯的。

关于游戏循环和有限态计算机(FSM)的内容将在本章最后涉及 FreakOut 演示游戏的章节中再进行更详细的讨论。

常规游戏编程指导

下面讨论一下读者所关心，也是游戏编程常用的技术和基本原理，这有利于简化游戏编程的复杂程度。

首先，视频游戏是运行于超高性能计算机上的游戏程序。对于时间或内存要求特别严格的代码部分不能使用高级 API 来编程，和游戏代码内部盾环有关的部分，大都需自己手工编写，否则游戏将会碰到严重的速度和性能问题。当然，这并不意味着就不能信任 DirectX 等 API 编程工具，因为 DirectX 以高性能和心可能“瘦”的方式编写。但在通常情况下，要避免高级的函数调用。

除上述情况应多加注意外，在编程时还应留意下面的编程技巧。

技巧：不要怕使用全局变量，许多视频游戏不使用大量的带有形参的、与时间相关的函数，而是使用一个全局变量来代替，例如一个函数的代码如下：

```
void Plot(int x,int y,int color)
{
  //在屏幕上画一个点像素
  video_buffer[x + y * MEMORY_PITCH] = color;
}//结束 Plot
```

函数体运行的时间小于函数调用所需的时间。这是由于参数压入和弹出堆栈造成的。在这种情况下，更好的方法可能是创建一个全局变量，然后在调用前进行赋值，像下面一样：

```
int gx,gy,gz,gcolor;//定义一些全局变量
```

```
void Plot_G(void)
{
  //使用全局变量来画一个点像素
  video_buffer(gx + gy * MEMORY_PITCH] = gcolor;
}//结束 Plot_G
```

技巧：使用内联功能。通过使用内联指令来完全摆脱调用功能甚至能够改善上面的技巧。内联指令不调用函数，而指示编译器将被调用函数代码放在需要调用该函数的最佳位置，这样做会使程序变得更大，但却提高了运行速度。下面是一个实例。

```
inline void Plot_I (int x,int y,int color)
{
  //在屏幕上画一个点像素
  video_buffer[x + y * MEMORY_PITCH] = color;
}//结束 Plot_I
```

注意：这里并没有使用全局变量，因为编辑器有效运行了相同类型数据的别名，但是全局变量迟早会派上用场，如果在函数调用时，一个或两个形参已改变，由于没有重新加载，所以旧的参数值有可能仍被使用。

技巧：尽量使用 32 位变量而不用 8 位变量或 16 位变量，Pentium 和之后的处理器全部都是 32 位的，这就意味着它们并不喜欢 8 位或 16 位的数据字符，实际上，更小的数据可能会由于超高速缓存和其他相关的内存寻址异常而使速度下降，例如，读者可能创建一个如下所示的结构：

```
struct CPOINT
{
  short x,y;
  unsigned char c;
}//结束 CPINT
```

尽管这个结构看上去很好，但实际并非如此！首先，结构本身目前是一个 5 字符长的结构——2 个 short+1 个 char=5。这实际上是一个很差的设计，这将导致内存地址崩溃。更好的结构形式如下：

```
struct CPOINT
{
  int x,y;
  int c;
}//结束 CPINT
```

Ｃ＋＋　提示：除了默认的公共可见性外，C++中的结构更像是“类”。

这种新结构要更好一些。首先，所有的成员都是相同尺寸——也就是说整数的大小为 4 字节。因此，单个指针可以通过递增 DWORD(双字节)的边界访问任何单元。当然，这种新结构的大小　就是 3 个整数长，即 12 字节，至少是 4 的倍数，或者在 DWORD 边界上。这样将明显地提升性能。

实际上，如果读者真想稳妥地话，应当将所有的结构都变为 32 字节的倍数。

由于 Pentium 家庭处理器芯片上标准缓存线长度是 32 倍数，因而这是一个最佳长度。可以通过人工虚设单元或者使用编辑指令(最简单的方法)来满足这个要求。当然，这可能会浪费大量的内存，但是为了提高速度这是值得的。

技巧：注释你的代码，游戏程序员不注释代码是出了名的，不要犯相同的错误。用额外的输入换取整洁，注释良好的代码是值得的。

技巧：程序应以类似 RISC(精简指令系统计算机)的形式来编写。换句话说，尽量简化你的代码，而不是使它更复杂。Pentium 和 PentiumⅡ处理器特别喜欢简单指令，而不是复杂的指令，你的程序可以长些，但应尽量使用简单指令，使程序相对于编辑器来说更加简单些。例如，不要编写类似下面的程序：

```
if((x+=(2*buffer[index++])>10)
{
  //进行工作
}//结束
```

应这样做

```
x+=(2*buffer[index]);
index++
if(x>10)
{
  //进行工作
}//结束 if
```

技巧：按照这种方式来编写代码有两个原因，首先，它允许调试程序在代码各部分之间放置断点;第二,这将更易于编译器向 Pentium 处理器传送简单指令，这样将使处理器使用更多的执行单元并行地处理代码。复杂的代码就比较糟糕。

对于简单的、是 2 的倍数的整数乘法运算，应使用二进制移位运算。因为所有的数据在计算机中都以二进制存储,位组合向左或右移动就等同于乘法和除法运算。例如：

```
int y_pos = 10
//将 y_pos 乘以 64
y_pos = (y_pos << 6);//2^6=64
```

相似的有

```
{
//将 y_pos 除以 8
y_pos = (y_pos >> 3);//1/2^3=1/8
```

当读者接触到优化那一章时，将会发现更多的、类似的技巧。哈哈，酷吧！

技巧：编写高效的算法。世界上所有的汇编语言都不会将 n^2 算法运行得更快些，更好的方法是使用整齐、高效的算法而不是蛮干。

技巧：不要在编程过程中优化代码。这通常会浪费时间。等到完成主要的代码块或整个程序后才开始进行繁重的的优化工作。这样可以节省你的时间，因为你必须处理一些模糊的代码或不必要的优化。当游戏编程完成时，才到了剖析代码、查找问题以优化程序的时间。另一方面，程序要注意错落有致，不要杂乱无章。

技巧：不要为简单的对象编写大量的复杂的数据结构，仅仅因为连接的清单非常酷并不意味着必须使用它们，对于静态数组而言，其元素一般为 256 个，的以只需为之静态分配内存并进行相应的处理即可。视频游戏编程 90%都是数据操作，游戏程序的数据应尽可能简单、可见，以便能够迅速地存取它、随意操作它或进行其他处理，确保你的数据结构按照这一原则进行处理。

技巧：使用 C++应谨慎。如果读者是位老练的高手，继续前进去做你喜欢的事，但是不要去疯狂追求类，或使游戏程序过于复杂以至于超出一般计算机的承受能力，简单、直观的代码是最好的程序，也最容易调试。我从来都不想看多重的隶属关系。

技巧：如果感到前路荆棘丛生，那就停下来，回头然后绕路而行，我见过许多游戏程序员开始于一条很差的编程路线，然后葬送自己。能够意识到自己所犯的错误，然后重新编写 500 行的代码要比得到一个不是期望的代码结构要好得多，因此，如果在工作中发现问题，那就要重新评估并确保它是值得花时间补救的。

技巧：经常备份你的工作。在编写游戏代码时，需要相当频繁地锁定系统。重新做一个排序算法比较容易，但是要为一个新角色或碰撞检测重新编写 AI 则是另一回事。

技巧：在开始你的游戏项目之前，进行一下组织工作，使用合理的文件名和目录名，提出一种一致的变量命名约定，尽量对图形和声音数据使用分开的目录，而不是将其全部都放置在一个目录中。

使用工具

过去编写视频游戏通常只不过需要一个文本编辑器和一个简略的自制图形程序。但是现在事情就变得复杂一点了，读者至少需要一个 C/C++编译器、一个 2D 的图形程序和一个声音处理程序。此外，如果读者想编写一个 3D 游戏的话，读者可能还需要一个 3D 的模型，而如果读者想使用任何 MIDI 设备的话，还要有一个音乐排序程序。

让我们来浏览一下目前流行的产品及其功用。

C/C++编译器

对于 Windows 9X/NT 的研制来讲，简直没有比 MS VC++5.0+更好的编译器了。它可以做任何读者想做的事，甚至更多。所产生的.EXE 文件是最快的有效代码。Borland 编译器也可以工作得很好(并且它要便宜得多)，但是它的特性设置较

少。在任何情况下，读者不一定需要上述任何一种编译器增强版本，一个能够产生 Win32 平台下的.EXE 文件的学生版本就已经足够了。

2D 艺术软件

这儿读者可以得到图形程序、画图程序和图像处理程序。图形程序主要允许读者以原色一个像素、一个像素地绘制和变换图形。直到现在为止，JASC 公司的 Paint Shop Pro5.0+还是性价比最佳的软件包。Fractal Design Painter 也很好，但是它更适用于传统的艺术家，而且它很昂贵。我喜欢使用 Corel Photo-Paint，但是对于网络游戏新手的需要来讲，它的功能的确有点偏大。

另一方面，画图程序允许读者创建图像，通常用曲线、直线和 2D 的几何原型来创建图像。这些程序并不是很有用，但如果读者需要的话，Adobe Illustrator 是一个很好的选择。

2D 艺术程序中的最后一类是图像处理类。这些程序多用于产品的后期制作，而不是前期的艺术创建。Adobe Photoshop 是大多数人喜欢的软件，但是我认为 Corel Photo-Paint 更好一些。读者自己来决定吧。

声音处理软件

目前用于游戏的所有的声音效果(SFX)90%都是数字样本，采用这种类型的声音数据来工作，读者应当需要一个数字声音处理程序。这一类中最好的程序是 Sound Forge Xp。目前为止，它具有我所见到的最复杂的声音处理能力，并且使用也最方便。

3D 造型软件

这是挑战经济实力的软件。3D 造型软件价格可能需要上万美金，但是最近也有大量的低价的 3D 造型软件上市，并且也具有足够的功能来制作一部影片。我主要是使用简单中等复杂程序的 3D 造型和动画软件——Caligari trueSpace Ⅲ+。在这种价位上，这是最好的 3D 造型软件，只要几百美金，并且拥有最好的界面。

如果读者希望功能更加强大并追求绝对的超级现实主义，3D Studio Max Ⅱ+就可以做到这一点，但是它的价格大约要 2500 美金，因此应当认真考虑一下。然而如果我们使用这些造型软件只是用来创建 3D 网络，而不是渲染，那么所有的其他修饰也就不需要了。这样 tureSpace 就足以应付。

音乐和 MIDI 排序程序

目前的游戏中有两类音乐:纯数字式(像 CD 一样)和 MIDI(乐器数字界面)式，MIDI 是一种基于人工记录数据的合成音效。如果想制作 MIDI 信息和歌曲，读者还需要一个排序软件包。一种性价比最佳的软件包是 Cakewalk，因此，如果读者打算记录和制作 MIDI 音乐的话，建议最好去了解一下这个软件。在涉及 DirectMusic 内容的第 10 章“用 DirectSound 和 DirectMusic 演奏乐曲中”中，我们将对 MIDI 数据再作探讨。

技巧：现在是最酷的部分.....许多软件制造商将许我在 CD 上列出了它们软件的共享版或评测版，赶快去体验这些软件吧！

从准备到完成——使用编译器

学习 Windows 游戏编程的一件最容易令人灰心丧气的事是学习如何使用编译器。大多数情况下，读者对开始编写游戏程序如此激动，以至于全身心地投入到 IDC（集成开发环境）中去尝试进行编译，然后就出现了一百万条编译和软件错误！为了有助于解决这个问题，让我们首先回顾一下有关编译器的一些基本概念。

0. 请读者务必阅读全部编译器指令！

1. 必须在你的系统上安装 DirectX SDK(软件开发工具包)。你所要做的就是在 CD 上查找到<DirectX SDK>上当，阅读 README.TXT 文件，并按说明进行操作(实际上只不过是“单击 DiretC SDK INSTALL.EXE 程序”)。

2. 我们要制作的是 Win32.EXE 程序，而不是.DLL 文件和 ActiveX 组件等等。因此如果读者想编译的话，需要做的第一件事情是使用编译器创建一个新的工程或工作区，然后将目标输出文件设定为 Win32.EXE。使用 VC++5.0 编译器进行这一步的工作如图 1.6 所示。

3. 应用添加文件(ADD Files)命令从主菜单或工程节点本身向工程添加源文件。对于使用 VC++5.0 编译器而言。其操作过程如图 1.7 所示。

4. 从接触到 DirectX 一章时起，就必须要包含下面列出的和图 1.8 所表示的 DirectX COM 界面库文件。

- DDRAW.LIB
- DSOUND.LIB
- DSOUND3D.LIB
- DINPUT.LIB
- DMUSIC.LIB
- DSETUP.LIB

这些 DirectX.LIB 文件位于所安装的 DirectX SDK 根目录下的<LIB>目录下。必须将这些.LIB 库文件添加到读者的工程或工作区中。读者不可能只添加搜索路径，因为搜索引擎会发现编译器本身安装的库文件和旧的 DirectX3.0 的.LIB 文件。如果是这样做的话，读者可能不得不将 Windows 多媒体扩展库文件——WINMM.LIB 加入到工程中去。该文件位于读者的编译器安装目录下的<LIB>目录下。

5. 准备编译你的程序

警告：如果读者是 Borland 用户，在 DirectX 软件开发工具包中有一个单独的 Borland 库文件目录，因此要确保将这些.LIB 文件而不目录树中上一级的 MS 兼容文件添加到工程中。

如果读者仍然对此有疑问的话，请不必着急。在本书中，当讨论 Windows 编程和首次接触 DirectX 时还要多次重复这些步骤。

实例：FreakOut

在沉溺于所讨论的有关 Windows、DirectX 和 3D 图形之前，应当轶一下，先显示一个完整的游戏——虽然简单了一点，但毫无疑问是一个完整的游戏。读者可以看到一个起初的游戏循环和一些图形的调用以及一些编码工作。怎么样？跟我来吧！

问题是我们现在仅仅在第一章。我并不喜欢使用后面章节中的内容.....这听起来欺骗读者，对吧？因此，我决定要做是使用以前常常使用的黑匣子 API 来进行游戏编程。基于这个要求，我要提一个问题“创建一个类似 Freakout 的 2D 游戏，其最低要求是什么？”我们真正所需要的是下面的功能：

- 转换为任何图形模式
- 在屏幕上画各种颜色的矩形
- 采用键盘输入
- 使用定时函数同步游戏循环
- 在屏幕上画一串带颜色的文本

因此我创建一个名字为 BLACKBOX。CPP|H 的目录。该程序带有一套 DirectX 函数集，并且包含实现所需要功能的支持代码。最妙的是，读者根本不需要去查看这些代码，只要在这些函数原型的基础上使用这些函数就可以了，并确保连接上 BLACKBOX.CPP|H 文件来产生一个.EXE 文件。

以 BLACKBOX 库为基础，我编写了一个名字为 FreakOut 的游戏，这个游戏演示了本章中所讨论的许多概念。FreakOut 游戏含有实际游戏的全部主要组成部分，包括：一个游戏循环、计分、等级，甚至还带有一个球的小型物理模型。我所说的是小型的模型。图 1.9 是一幅游戏运行中的屏幕画面。当然它远远不及 Arkanoid，但 4 个小时的工作有此成果也不赖！

在编写游戏代码之前，我希望读者能看一下工程和其各种构件是如何协调一致的。参见图 1.10

从图中可以看到，游戏主要由下面文件构成：

FREAKOUT.CPP——主要的游戏逻辑，使用 BLACKBOX.CPP，创建一个最小的 Win32 应用程序。

BLACKBOX.CPP——游戏库(请不要偷看)

BLACKBOX.H——游戏库头文件。

DDRAW.LIB——用来建立应用程序的 DriectDraw 输入库。其中并不含有真正的 DirectX 代码。主要用来作为一个函数调用的中间库，可以轮流调用进行实际工作 DDRAW.DLL 动态链接库。它可以在 DirectX SDK 安装程序<LIB>目录下找到。

DDRAW.DLL——运行过程中的 DirectDraw 库，实际上含有通过 DDRAW.LIB 输入库调用 DirectDraw 界面函数的 COM 执行程序。对此读者不必担心：只要确认已经安装了 DirectX 文件即可。

现在，我们对此已有了了解，让我们看一下 BLACKOUT.H 头文件，看看它包含了哪些函数。

程序清单 1.2 BLOCKOUT.H 头文件

```
-------------------------
// BLACKBOX.H - Header file for demo game engine library
```

```
// watch for multiple inclusions
#ifndef BLACKBOX
#define BLACKBOX

// DEFINES ////////////////////////////////////////////////

// default screen size
#define SCREEN_WIDTH 640 // size of screen
#define SCREEN_HEIGHT 480
#define SCREEN_BPP 8 // bits per pixel
#define MAX_COLORS 256 // maximum colors

// MACROS /////////////////////////////////////////////////

// these read the keyboard asynchronously
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// initializes a direct draw struct
#define DD_INIT_STRUCT(ddstruct)
{ memset(&ddstruct,0,sizeof(ddstruct));
ddstruct.dwSize=sizeof(ddstruct); }

// TYPES //////////////////////////////////////////////////

// basic unsigned types
typedef unsigned short USHORT;
typedef unsigned short WORD;
typedef unsigned char UCHAR;
typedef unsigned char BYTE;

// EXTERNALS //////////////////////////////////////////////

extern LPDIRECTDRAW4 lpdd; // dd object
extern LPDIRECTDRAWSURFACE4 lpddsprimary; // dd primary surface
extern LPDIRECTDRAWSURFACE4 lpddsback; // dd back surface
extern LPDIRECTDRAWPALETTE lpddpal; // a pointer to the created dd palette
extern LPDIRECTDRAWCLIPPER lpddclipper; // dd clipper
extern PALETTEENTRY palette[256]; // color palette
extern PALETTEENTRY save_palette[256]; // used to save palettes
extern DDSURFACEDESC2 ddsd; // a direct draw surface description struct
extern DDBLTFX ddbltfx; // used to fill
extern DDSCAPS2 ddscaps; // a direct draw surface capabilities struct
extern HRESULT ddrval; // result back from dd calls
extern DWORD start_clock_count; // used for timing
```

```
// these defined the general clipping rectangle
extern int min_clip_x, // clipping rectangle
max_clip_x,
min_clip_y,
max_clip_y;

// these are overwritten globally by DD_Init()
extern int screen_width, // width of screen
screen_height, // height of screen
screen_bpp; // bits per pixel

// PROTOTYPES ////////////////////////////////////////////

// DirectDraw functions
int DD_Init(int width, int height, int bpp);
int DD_Shutdown(void);
LPDIRECTDRAWCLIPPER DD_Attach_Clipper(LPDIRECTDRAWSURFACE4 lpdds, int
num_rects, LPRECT clip_list);
int DD_Flip(void);
int DD_Fill_Surface(LPDIRECTDRAWSURFACE4 lpdds,int color);

// general utility functions
DWORD Start_Clock(void);
DWORD Get_Clock(void);
DWORD Wait_Clock(DWORD count);

// graphics functions
int Draw_Rectangle(int x1, int y1, int x2, int y2, int
color,LPDIRECTDRAWSURFACE4 lpdds=lpddsback);

// gdi functions
int Draw_Text_GDI(char *text, int x,int y,COLORREF color,
LPDIRECTDRAWSURFACE4 lpdds=lpddsback);
int Draw_Text_GDI(char *text, int x,int y,int color,
LPDIRECTDRAWSURFACE4 lpdds=lpddsback);

#endif
```

现在，就不要将你的精力都浪费在程序代码和那些神秘的全局变量是什么的问题上，让我们来看一看这些函数。如读者所看到一样，这个简单的图形界面需要一些函数来完成。在这个图形和小型的 Win32 应用程序（我们要做的 Windows 编程工作越少越好）的基础上，我创建了游戏 FREAKOUT.CPP 如实例 1.1 所示。请认真地看一看，特别是主游戏循环和对游戏处理功能的访问。

程序清单 1.3 FREAKOUT.CPP 源文件

```
---------------------------
// FREAKOUT.CPP - break game demo

// INCLUDES ///////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // include all macros
#define INITGUID // include all GUIDs

#include <windows.h> // include important windows stuff
#include <windowsx.h>
#include <mmsystem.h>

#include <iostream.h> // include important C/C++ stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h> // directX includes
#include "blackbox.h" // game library includes

// DEFINES ////////////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WIN3DCLASS" // class name

#define WINDOW_WIDTH 640 // size of window
#define WINDOW_HEIGHT 480

// states for game loop
#define GAME_STATE_INIT 0
#define GAME_STATE_START_LEVEL 1
#define GAME_STATE_RUN 2
#define GAME_STATE_SHUTDOWN 3
#define GAME_STATE_EXIT 4
```

```
// block defines
#define NUM_BLOCK_ROWS 6
#define NUM_BLOCK_COLUMNS 8

#define BLOCK_WIDTH 64
#define BLOCK_HEIGHT 16
#define BLOCK_ORIGIN_X 8
#define BLOCK_ORIGIN_Y 8
#define BLOCK_X_GAP 80
#define BLOCK_Y_GAP 32

// paddle defines
#define PADDLE_START_X (SCREEN_WIDTH/2 - 16)
#define PADDLE_START_Y (SCREEN_HEIGHT - 32);
#define PADDLE_WIDTH 32
#define PADDLE_HEIGHT 8
#define PADDLE_COLOR 191

// ball defines
#define BALL_START_Y (SCREEN_HEIGHT/2)
#define BALL_SIZE 4

// PROTOTYPES ////////////////////////////////////////////////

// game console
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// GLOBALS ///////////////////////////////////////////////////

HWND main_window_handle = NULL; // save the window handle
HINSTANCE main_instance = NULL; // save the instance
int game_state = GAME_STATE_INIT; // starting state

int paddle_x = 0, paddle_y = 0; // tracks position of paddle
int ball_x = 0, ball_y = 0; // tracks position of ball
int ball_dx = 0, ball_dy = 0; // velocity of ball
int score = 0; // the score
int level = 1; // the current level
int blocks_hit = 0; // tracks number of blocks hit

// this contains the game grid data

UCHAR blocks[NUM_BLOCK_ROWS][NUM_BLOCK_COLUMNS];
```

```
// FUNCTIONS ////////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
UINT msg,
WPARAM wparam,
LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT ps; // used in WM_PAINT
HDC hdc; // handle to a device context

// what is the message
switch(msg)
{
case WM_CREATE:
{
// do initialization stuff here
return(0);
} break;

case WM_PAINT:
{
// start painting
hdc = BeginPaint(hwnd,&ps);

// the window is now validated

// end painting
EndPaint(hwnd,&ps);
return(0);
} break;

case WM_DESTROY:
{
// kill the application
PostQuitMessage(0);
return(0);
} break;

default:break;

} // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));
```

```
} // end WinProc

// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain( HINSTANCE hinstance,
HINSTANCE hprevinstance,
LPSTR lpcmdline,
int ncmdshow)
{
// this is the winmain function

WNDCLASS winclass; // this will hold the class we create
HWND hwnd; // generic window handle
MSG msg; // generic message
HDC hdc; // generic dc
PAINTSTRUCT ps; // generic paintstruct

// first fill in the window class stucture
winclass.style = CS_DBLCLKS | CS_OWNDC |
CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = WINDOW_CLASS_NAME;

// register the window class
if (!RegisterClass(&winclass))
return(0);

// create the window, note the use of WS_POPUP
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
"WIN3D Game Console", // title
WS_POPUP | WS_VISIBLE,
0,0, // initial x,y
GetSystemMetrics(SM_CXSCREEN), // intial width
GetSystemMetrics(SM_CYSCREEN), // initial height
NULL, // handle to parent
NULL, // handle to menu
hinstance,// instance
```

```
NULL))) // creation parms
return(0);

// hide mouse
ShowCursor(FALSE);

// save the window handle and instance in a global
main_window_handle = hwnd;
main_instance = hinstance;

// perform all game console specific initialization
Game_Init();

// enter main event loop
while(1)
{
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
{
// test if this is a quit
if (msg.message == WM_QUIT)
break;

// translate any accelerator keys
TranslateMessage(&msg);

// send the message to the window proc
DispatchMessage(&msg);
} // end if

// main game processing goes here
Game_Main();

} // end while

// shutdown game and release all resources
Game_Shutdown();

// show mouse
ShowCursor(TRUE);

// return to Windows like this
return(msg.wParam);

} // end WinMain
```

```
// T3DX GAME PROGRAMMING CONSOLE FUNCTIONS ////////////////////

int Game_Init(void *parms)
{
// this function is where you do all the initialization
// for your game


// return success
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// this function is where you shutdown your game and
// release all resources that you allocated


// return success
return(1);

} // end Game_Shutdown

//////////////////////////////////////////////////////////////

void Init_Blocks(void)
{
// initialize the block field
for (int row=0; row < NUM_BLOCK_ROWS; row++)
for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
blocks[row][col] = row*16+col*3+16;

} // end Init_Blocks

//////////////////////////////////////////////////////////////

void Draw_Blocks(void)
{
// this function draws all the blocks in row major form
int x1 = BLOCK_ORIGIN_X, // used to track current position
y1 = BLOCK_ORIGIN_Y;
```

```
// draw all the blocks
for (int row=0; row < NUM_BLOCK_ROWS; row++)
{
// reset column position
x1 = BLOCK_ORIGIN_X;

// draw this row of blocks
for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
{
// draw next block (if there is one)
if (blocks[row][col]!=0)
{
// draw block
Draw_Rectangle(x1-4,y1+4,
x1+BLOCK_WIDTH-4,y1+BLOCK_HEIGHT+4,0);

Draw_Rectangle(x1,y1,x1+BLOCK_WIDTH,
y1+BLOCK_HEIGHT,blocks[row][col]);
} // end if

// advance column position
x1+=BLOCK_X_GAP;
} // end for col

// advance to next row position
y1+=BLOCK_Y_GAP;

} // end for row

} // end Draw_Blocks

//////////////////////////////////////////////////////////

void Process_Ball(void)
{
// this function tests if the ball has hit a block or the paddle
// if so, the ball is bounced and the block is removed from
// the playfield note: very cheesy collision algorithm :)

// first test for ball block collisions

// the algorithm basically tests the ball against each
// block's bounding box this is inefficient, but easy to
// implement, later we'll see a better way
```

```
int x1 = BLOCK_ORIGIN_X, // current rendering position
y1 = BLOCK_ORIGIN_Y;

int ball_cx = ball_x+(BALL_SIZE/2), // computer center of ball
ball_cy = ball_y+(BALL_SIZE/2);

// test of the ball has hit the paddle
if (ball_y > (SCREEN_HEIGHT/2) && ball_dy > 0)
{
// extract leading edge of ball
int x = ball_x+(BALL_SIZE/2);
int y = ball_y+(BALL_SIZE/2);

// test for collision with paddle
if ((x >= paddle_x && x <= paddle_x+PADDLE_WIDTH) &&
(y >= paddle_y && y <= paddle_y+PADDLE_HEIGHT))
{
// reflect ball
ball_dy=-ball_dy;

// push ball out of paddle since it made contact
ball_y+=ball_dy;

// add a little english to ball based on motion of paddle
if (KEY_DOWN(VK_RIGHT))
ball_dx-=(rand()%3);
else
if (KEY_DOWN(VK_LEFT))
ball_dx+=(rand()%3);
else
ball_dx+=(-1+rand()%3);

// test if there are no blocks, if so send a message
// to game loop to start another level
if (blocks_hit >= (NUM_BLOCK_ROWS*NUM_BLOCK_COLUMNS))
{
game_state = GAME_STATE_START_LEVEL;
level++;
} // end if

// make a little noise
MessageBeep(MB_OK);

// return
return;
```

```
} // end if

} // end if

// now scan thru all the blocks and see of ball hit blocks
for (int row=0; row < NUM_BLOCK_ROWS; row++)
{
// reset column position
x1 = BLOCK_ORIGIN_X;

// scan this row of blocks
for (int col=0; col < NUM_BLOCK_COLUMNS; col++)
{
// if there is a block here then test it against ball
if (blocks[row][col]!=0)
{
// test ball against bounding box of block
if ((ball_cx > x1) && (ball_cx < x1+BLOCK_WIDTH) &&
(ball_cy > y1) && (ball_cy < y1+BLOCK_HEIGHT))
{
// remove the block
blocks[row][col] = 0;

// increment global block counter, so we know
// when to start another level up
blocks_hit++;

// bounce the ball
ball_dy=-ball_dy;

// add a little english
ball_dx+=(-1+rand()%3);

// make a little noise
MessageBeep(MB_OK);

// add some points
score+=5*(level+(abs(ball_dx)));

// that's it -- no more block
return;

} // end if

} // end if
```

```
// advance column position
x1+=BLOCK_X_GAP;
} // end for col

// advance to next row position
y1+=BLOCK_Y_GAP;

} // end for row

} // end Process_Ball

//////////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// this is the workhorse of your game it will be called
// continuously in real-time this is like main() in C
// all the calls for you game go here!

char buffer[80]; // used to print text

// what state is the game in?
if (game_state == GAME_STATE_INIT)
{
// initialize everything here graphics
DD_Init(SCREEN_WIDTH, SCREEN_HEIGHT, SCREEN_BPP);

// seed the random number generator
// so game is different each play
srand(Start_Clock());

// set the paddle position here to the middle bottom
paddle_x = PADDLE_START_X;
paddle_y = PADDLE_START_Y;

// set ball position and velocity
ball_x = 8+rand()%(SCREEN_WIDTH-16);
ball_y = BALL_START_Y;
ball_dx = -4 + rand()%(8+1);
ball_dy = 6 + rand()%2;

// transition to start level state
game_state = GAME_STATE_START_LEVEL;
```

```
} // end if
////////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_START_LEVEL)
{
// get a new level ready to run

// initialize the blocks
Init_Blocks();

// reset block counter
blocks_hit = 0;

// transition to run state
game_state = GAME_STATE_RUN;

} // end if
////////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_RUN)
{
// start the timing clock
Start_Clock();

// clear drawing surface for the next frame of animation
Draw_Rectangle(0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1,200);

// move the paddle
if (KEY_DOWN(VK_RIGHT))
{
// move paddle to right
paddle_x+=8;

// make sure paddle doesn't go off screen
if (paddle_x > (SCREEN_WIDTH-PADDLE_WIDTH))
paddle_x = SCREEN_WIDTH-PADDLE_WIDTH;

} // end if
else
if (KEY_DOWN(VK_LEFT))
{
// move paddle to right
paddle_x-=8;

// make sure paddle doesn't go off screen
```

```
if (paddle_x < 0)
paddle_x = 0;

} // end if

// draw blocks
Draw_Blocks();

// move the ball
ball_x+=ball_dx;
ball_y+=ball_dy;

// keep ball on screen, if the ball hits the edge of
// screen then bounce it by reflecting its velocity
if (ball_x > (SCREEN_WIDTH - BALL_SIZE) || ball_x < 0)
{
// reflect x-axis velocity
ball_dx=-ball_dx;

// update position
ball_x+=ball_dx;
} // end if

// now y-axis
if (ball_y < 0)
{
// reflect y-axis velocity
ball_dy=-ball_dy;

// update position
ball_y+=ball_dy;
} // end if
else
// penalize player for missing the ball
if (ball_y > (SCREEN_HEIGHT - BALL_SIZE))
{
// reflect y-axis velocity
ball_dy=-ball_dy;

// update position
ball_y+=ball_dy;

// minus the score
score-=100;
```

```
} // end if

// next watch out for ball velocity getting out of hand
if (ball_dx > 8) ball_dx = 8;
else
if (ball_dx < -8) ball_dx = -8;

// test if ball hit any blocks or the paddle
Process_Ball();

// draw the paddle and shadow
Draw_Rectangle(paddle_x-8, paddle_y+8,
paddle_x+PADDLE_WIDTH-8,
paddle_y+PADDLE_HEIGHT+8,0);

Draw_Rectangle(paddle_x, paddle_y,
paddle_x+PADDLE_WIDTH,
paddle_y+PADDLE_HEIGHT,PADDLE_COLOR);

// draw the ball
Draw_Rectangle(ball_x-4, ball_y+4, ball_x+BALL_SIZE-4,
ball_y+BALL_SIZE+4, 0);
Draw_Rectangle(ball_x, ball_y, ball_x+BALL_SIZE,
ball_y+BALL_SIZE, 255);

// draw the info
sprintf(buffer,"F R E A K O U T Score %d Level %d",score,level);
Draw_Text_GDI(buffer, 8,SCREEN_HEIGHT-16, 127);

// flip the surfaces
DD_Flip();

// sync to 33ish fps
Wait_Clock(30);

// check of user is trying to exit
if (KEY_DOWN(VK_ESCAPE))
{
// send message to windows to exit
PostMessage(main_window_handle, WM_DESTROY,0,0);

// set exit state
game_state = GAME_STATE_SHUTDOWN;

} // end if
```

```
} // end if
//////////////////////////////////////////////////////////
else
if (game_state == GAME_STATE_SHUTDOWN)
{
// in this state shut everything down and release resources
DD_Shutdown();

// switch to exit state
game_state = GAME_STATE_EXIT;

} // end if

// return success
return(1);

} // end Game_Main

//////////////////////////////////////////////////////////
```

哈哈，酷吧？这是一个完整的 Wind32/DirectX 游戏，表现接近上佳水平了。BLACKOUT.CPP 源文件中有几百行代码，但是我们只能将之视为类 DirectX 游戏而且是由某个人编写的（我！）。不管怎样说，还是让我们迅速浏览一下实例 1.3 的内容吧。

首先，Windows 需要一个事件循环。这就是 Windows 编程的标准，因为 Windows 在大部分情况下都是事件驱动的。但是游戏却不是事件驱动的。无论用户在干什么，它们都在一直运行。因此，我们至少需要支持小型事件循环以取悦于 Windows。执行这项功能的代码位于 WinMain()中——呀，令人惊奇！不是吗？

WinMain()是所有 Windows 程序的主要入口点，就像是 Main()是所有 DOS/UNIX 程序中的入口点一样。任何情况下，FREAOUT 的 WinMain()创建一个窗口，正确进入到事件循环中。如果 Windows 需要工作时，就按照这样做。当所有的基本事件处理都结束时，访问 Game_Main()。这就是我们的游戏程序实际运行的状态。

如果需要的话，读者可以一直在 Game_Main()中循环。而不释放回到 WinMain()主事件循环体中。但这样做不是件好事，因为 Windows 会得不到任何信息，从而缺乏资源。哎，我们想要做的是运行一个动画和逻辑的画面，然后返回到 WinMain()。这样的话，Windows 可以继续工作和处理信息。如果所有这些听起来像是幻术的话，请不要担心——在下一章中情况还会更糟。

一旦进入 Game_Main()，就执行 FreakOut 逻辑。游戏图像提交到显示缓冲区，最后通过 DD_FLIP()访问在循环结束时在显示屏上显示出来。因此我希望读者要做的是浏览一下全部的游戏状态，跟随一遍游戏循环的每一部分，了解一下工作原理。要想玩游戏的话，只要单击 FREAKOUT.EXE 文件，程序就会立即启动。控制部分包括：

右箭头键——向右移动操作杆。

左箭头键——向左移动操作杆。

Esc 键——结束并返回到 Windows。

还有，如果读者错过一个球的话，将被罚掉 100 点，可要仔细盯紧啊！

如果读者对游戏代码和玩法已适应，试着修改一下游戏。可以增加不同的背景颜色(0～255 是各种有效的颜色)、更多的球、可以改变操作杆的大小以及更多的声音效果（那是我用 Win32API 功能 MessageBeep()函数做出的）。

总结

这大概是我所介绍的关于游戏编程快速入门课程中最快的一次了！我们谈论了大量的基础内容，但是只能把它看作本书的基础版本。我只想让读者对本书中我们所讨论的和学习的内容有一个感性认识。另外，阅读一个完整的游戏是有益的，因为这可以让读者思考若干问题。

现在，在进行关于 Windows 编程的第二章之前，读者对编译 FreakOut 游戏应该能够轻松驾驭。如果还没有这种感觉的话，就请立即打开编译器的书和 RTFM。我等着你们。

第二章 Windows 编程模型

Windows 编程就像去看牙科医生：明知道对你有好处，但就是没有人乐意去。是不是这样？在本章中，我将要使用“禅”的方法——或者换句话说，就是深入浅出地向你介绍 Windows 编程基础。我可不能保证在阅读了本章后你就会“去见牙科医生”，但是我也保证你会比以往更喜欢 Windows 编程。下面是本章的内容：

- Windows 的历史
- Windows 的基本风格
- Windows 的类
- 创建 Windows
- Windows 事件句柄
- 事件驱动编程和事件循环
- 打开多个窗口

Windows 的历史

读者可能会因为我要解放你的思想而感到十分恐惧(特别是钟情于 DOS 的顽固分子)。让我们迅速浏览一下 Windows 的发展历程以及与游戏发展的关系，好吗？

早期的 Windows 版本

Windows 的发展始于 Windows1.0 版本。这是 Microsoft 公司在商业视窗操作系统的第一次尝试，当然是一个非常失败的产品。Windows1.0 完全建立在 DOS 基础上(这就是一个错误)，不能执行多任务，运行速度很慢，看上去也差劲。它的外观可能是其失败的最重要原因。除了讽刺以外，问题还在于 Windows1.1 与那个时代的 80286 计算机(或更差劲的 8086)所能提供的相比需要更高的硬件、图像和声音性能。

然而，Microsoft 稳步前进，很快就推出了 Windows2.0。我记得获得 Windows2.0 的测试版时我正在软件出版公司工作。在会议室中，挤满了公司的行政官员和董事长(像往常一样，他正拿着一不鸡尾酒)。我们运行 Windows 2.0 测试演示版，装载了多个应用程序，看上去似乎还在工作。但是，那 IMB 推出了 PM。PM 看上去要好得多，它是建立在比 Windows2.0 先进得多的操作系统 OS/2 的基础上的，而 Windows 2.0 依然是建立在 DOS 基础上的视窗管理器。那天会议室中的结论是“不错，但还不是一个可行的操作系统，如果我们仍然留恋在 DOS 上，那我还能有鸡尾酒喝吗？”

Windows 3.x

1990 年，终于发生了翻天覆地的变化，因为 Windows 3.0 出世了，而且其表现的确非常出色！尽管它仍然赶不上 Mac OS 的标准，但是谁还在意呢？(真正的程序员都憎恨 Mac)。软件开发人员终于可以在 PC 机上创建迷人的应用程序了，而商用应用程序也开始脱离 DOS。这成了 PC 机的转折点，终于将 Mac 完全排除在商用程序之外了，而后也将其挤出台式机出版业。(那时，Apple 公司每 5 分钟就推出一种新硬件)。

尽管 Windows3.0 工作良好，却还是存在许多的问题、软件漏洞，但从技术上说它已是 Windows2.0 之后的巨大突破，有问题也是在所难免。为了解决这些问题，Microsoft 推出了 Windows3.1，开始公关部和市场部打算称之为 Windows4.0，但是，Microsoft 决定只简单地称之为 Windows3.1，因为它不足以称之为升级的换代版本。它还没有做到市场部广告宣传的那样棒。

Windows3.1 非常可靠。它带有多媒体扩展以提供音频和视频支持，而且它还是一个出色的、全面的操作系统，用户能够以统一的方式来工作。另外，还存在一些其他版本。如可以支持网络的 Windows3.11(适用于工作的 Windows)。唯一的问题是 Windows3.1 仍然是一个 DOS 应用程序，运行 DOS 扩展器上。

Windows95

另一方面，游戏编程行业还在唱“DOS 永存！”的赞歌，而我则已经开始热衷于使用 Windows3.1。但是，1995 年世界开始冷却——Windows95 终于推出。它是一个真正 32 位的、多任务、多线程的操作系统。诚然，其中还残存一些 16 位代码，但在极大程度上，Windows95 是 PC 机的终极度开发和发布平台。

(当然，Windows NT 3.0 也同时推出，但是 NT 对于大多数用户来讲还是不可用的，因此这里也就不再赘述)。

当 Windows95 推出后，我才真正开始喜欢 Windows 编程。我一直痛恨使用 Windows1.0、2.0、3.0 和 3.1 来编程，尽管随着每一种版本的推出，这种憎恨都越来越少。当 Windows95 出现时，它彻底改变了我的思想，如同其他被征服的人的感觉一样——它看上去非常酷！那正是我所需要的。

提示：游戏编程行业中最重要的事情是游戏表现如何，游戏的画面如何，同时还要尽可能减轻审阅人的工作。

因此几乎一夜间，Windows95 就改变了整个计算机行业。的确，目前还有一些公司仍然在使用 Windows3.1(你能相信吗？)，但是 Windows95 使得基于 Intel 的 PC 成为除游戏之外　的所有应用程序的选择。不错，尽管游戏程序员知道 DOS 退出游戏编程行业只是个时间的问题了，但是 DOS 还是它们的核心。

1996 年，Microsoft 公司发布了 Game SDK(游戏软件开发工具包)，这基本上就是 DirectX 的第一个版本。这种技术仅能在 Window95 环境下工作，但是它的确太慢了，甚至竞争不过 DOS 游戏(如 DOOM 和 Duke Nukem 等)。游戏开发人员继续使用 DOS32 来开发游戏，但是他们知道 DirectX 具有足够快的速度，从而能使游戏能够流畅地运行在 PC 机上已为时不远。

到了 3.0 版，DirectX 的速度在同样计算机上已经和 DOS32 一样快了。到了 5.0 版，DirectX 已经相当完美，实现了该技术最初的承诺。现在要意识到：Win32/DirectX 是 PC 机上开发游戏的唯一方式。这是历史的选择。

Windows 98

1998 年中期，Windows 98 推出了。这至多是一个改进的版本，而不像 Windows95 那样是一个换代的产品，但毫无疑问它也占有很重要的地位。Windows98 像一辆旧车改装的高速汽车——实际上它是一头皮毛圆润光滑、脚力持久、飞奔跳动的驴。它是全 32 位的，能够支持你想做的所有事情，并具有无限扩充能力。它很好地集成了 DirectX、3D 图形、网络以及 Internet。

Windows98 和 Window95 相比也非常稳定。当然 Windows98 仍然经常死机，

但可以相信的是，这比 Windows95 少了许多。对即插即用支持得很好，并且能够很好地运行——这只是个时间问题。

Windows NT

现在我们来讨论一下 Windows NT。在本书编写期间，Windows NT 正在推出 5.0 版本。我所能说的是，它最终将取代 Windows 9X 成为每个人的操作系统选择。NT 要比 Windows9X 严谨得多；而且绝大多数游戏程序员都将在 NT 上开发游戏，这将使 Windows 9X 退出历史舞台。Windows NT 5.0 最酷的是它完全支持即插即用和 Win32/DirectX，因此使用 DirectX 为 Windows 9X 编写的应用程序可以在 WindowsNT5.0 或更高版本上运行。这可是个好消息。因为从历史上看，编写 PC 游戏的开发人员现在具有最大的市场潜力。

那么最低标准是什么呢？如果你使用 DirectX(或其他工具)编写了一个 Win32 应用程序，它完全可以在 Windows 95、98、和 NT 5.0 或更高版本上运行。这可是件好事情。因此你在本书中所学到的任何东西至少可以应用到三种操作系统上，也可以运行于安装 NT 和 DirectX 的其他计算机上，如 DEC 的 Alphas。还有 Windows CE——DirectX 和 Win32 衍生的运行于其他系统上的操作系统。

Windows 的基本风格：Win9X/NT

和 DOS 不同，Windows 是一个多任务的操作系统，允许许多应用程序和更小的程序同时运行，可以最大限度的发挥硬件的性能。这表明 Windows 是一个共享的环境——一个应用程序不可能独占整个系统。尽管 Windows 95、98、和 NT 很相似，但仍然存在许多技术上的差别。但是就我们所涉及的，不可能去详细归纳。这里所参照的 Windows 机器一般是指 Win9X/NT 或 Windows 环境。让我们开始吧！

多任务和多线程

如我所说，Windows 允许不同的应用程序以轮流的方式同时执行，每一个应用程序都占用一段很短的时间段来运行，下一个应用程序轮换运行。如图 2.1 所示，CPU 由几个不同的应用程序以轮流的方式共享。判断出下一个运行的应用程序、分配给每个应用程序的时间量是调度程序的工作。

调度程序可以非常简单——每个应用程序分配固定的运行时间，也可以非常复杂——将应用程序设定为不同的优先级和抢先性或低优先级的事件。就 WinX/NT 而言，调度程序采用基于优先级的抢先占用方式。这就意味着一些应用程序要比其他的应用程序占用处理器更多的时间，但是如果一个应用程序需要 CPU 处理的话，在另一任务运行的同时，当前的任务可以被锁定或抢先占用。

但是不要对此有太多的担心，除非你正在编写 OS(操作系统)或实时代码——其细节事关重大。大多数情况下，Windows 将执行和调度你的应用程序，无需你参与。

深入接触 Windows，我们可以看到，它不仅是多任务的，而且还是多线程的。这意味着程序由许多理简单的多个执行线程构成。这些线程(像更重要的进程)如程序一样被调度。实际上，在你的计算机上可同时运行 30～50 个线程，执行不同的任务。所以事实上你可能只运行一个程序，但这个程序由一个或多个执行线程构成。

Windows 实际的多线程示意图如图 2.2 所示，从图中可以看到，每一个程序实际上都是由一个主线程和几个工作线程构成。

获取线程的信息

下面让我们来看一下你的计算机现在正在运行多少个线程。在 Windows 机器上，同时按 Ctrl+ALt+Delete 键，弹出显示正在运行的任务(过程)的当前程序任务管理器。这和我们所希望的不同，但也很接近。我们希望的是一个显示正在执行的实际线程数的工具或程序。许多共享软件和商用软件工具都能做到这一点，但是 Windows 内嵌了这几个工具。

在安装 Windows 的目录(一般是 Windows)下，可以发现一个名字为 SYSMON.EXE(Windows 95/98)或 PREFMON.EXE 程序。图中除了正在运行的线程外还有大量的信息，如：内存使用和处理器装载等。实际上在进行程序开发时，我喜欢使 SYSMON.EXE 运行，由此可以了解正在进行什么以及系统如何加载程序。

你可能想知道能否对线程的创建进行控制，答案是能够！！！实际上这是 Windows 游戏编程最令人激动的事情之一——就像我们所希望的那样除了游戏主进程外，还能够执行其他的任务，我们也能够创建像其他任务一样多的线程。

注意：在 Windows98/NT 环境下，实际上还有一种叫 fiber 的新型执行对象，它比线程还简单(明白吗？线程是由 fiber 构成的)。

这和 DOS 游戏程序的编写有很大不同。DOS 是单线程操作系统，也就是说一旦你的程序开始运行，就只能运行该程序(不时出现的中断管理除外)。因此，如果想使用任何一种多任务或多线程，就必须自己来模拟(参阅《Sams Teach Yourself Game Programming in 21 Days》中关于一个完整的基于 DOS 多任务核心部分)。这也正是游戏程序员在这么多年中所作的事。的确，模拟多任务和多线程远远不能和拥有一个完整的支持多任务和多线程的操作系统相提并论。但是对于单个游戏来讲，它足可以良好地工作。

在我们接触到真正的 Windows 编程和那些工作代码之前，我想提及一个细节。你可能在想，Windows 真是一个神奇的操作系统，因为它允许多个任务和程序立即执行。请记住，实际上并不是这样的。如果不有一个处理器的话。那么一次也只能执行一个执行流，线程、程序或你所调用的任何对象。Windows 相互之间的切换太快了，以至于看上去就像几个程在同时运行一样。另一方面，如果有几个处理器的话，可以同时运行多个程序。例如，我有一个双 CPU 的 PentiumⅡ处理器在运行 WindowsNT5.0。使用这种配置，可以同时执行两个指令流。

我希望在不远的将来，个人计算机的新型微处理器结构能够允许多个线程或 fiber 同时执行，将这样一个目标作为处理器设计的一部分。例如，Pentium 具有两个执行单元——U 管和 V 管。因此它能够立即执行两个指令。但是，这两个指令都是来自同一个线程，类似的是 PentiumⅡ能够立即执行 5 个简单的指令，但也是来自同一个线程。

事件模型

Windows 是个多任务/多线程的操作系统，并且还是一个事件驱动的操作系统。和 DOS 程序不同的是，Windows 程序都是等着用户去使用，由此而触发一个

事件，然后 Windows 对该事件发生响应，进行动作。请看图 2.4 所示的示意图，图中描述了大量的应用程序窗口，每个程序都向 Windows 发送待处理的事件和消息。Windows 对其中的一些进行处理，大部分的消息和事件被传递给应用程序来处理。

这样做的好处是你不必去关心其他的正在运行的应用程序，Windows 会为你处理它们。你所要关心的就是你自己的应用程序和窗口中信息的处理。这在 Windows3.0/3.1 中是根本不可能的。Windows 的那些版本并不是真正的多任务操作系统，每一个应用程序都要产生下一个程序，也就是说，在这些版本的 Windows 下运行的应用程序感觉相当粗糙、缓慢。如果有其他应用程序干扰系统的话，这个正在“温顺地”运行的程序将停止工作。但这种情况在 Windows9X/NT 下就不会出现。操作系统将会适当的时间终止你的应用程序——当然，运行速度非常快，你根本就不会注意到。

到目前为止，读者已了解了所有有关操作系统的概念。幸运的是，有了 Windows 这个目前最好的编写游戏操作系统，读者根本就不必担心程序调度——你所要考的就是游戏代码和如何最大限度地发挥计算机的性能。

在本章后面内容中，我们要接触一些实际的编程工作，便于读者了解 Windows 编程有多容易。但是(永远都有但是)在进行实际编程之前我们应当了解一些 Microsoft 程序员喜欢使用的约定。这样你就不会被那些古怪的函数和变量名弄得不知所措。

按照 Microsoft 方式编程：匈牙利符号表示法

如果你正在动作一个像 Microsoft 一样的公司，有几千个程序员都在干不同的项目，在某一点上就应当提出一个编写代码的标准方式。否则，结果将是一片混乱。因此一个名字叫 Charles Simonyi 的人被委托创立了一套编写 Microsoft 代码的规范。这个规范已经用作编写代码的基本指导说明书。所有 Microsoft 的 API、界面、技术文件等等都采用这些规范。

这个规范通常被称为匈牙利符号表示法，可能是因为创立这个规范工作很长时间，弄得他饥肠辘辘的原因吧(英文中饥饿和匈牙利谐音)，或者可能他是匈牙利人。对我们根本不知道，关键是你必须了解这个规范，以便于你能够阅读 Microsoft 代码。匈牙利符号表示法包括许多与下列命名有关的约定：

- 变量
- 函数
- 类型和常量
- 类
- 参数

表 2.1 给出了匈牙利符号表示法使用的前缀代码。这些代码在大多数情况下一半用于前缀变量名，其他约定根据名称确定。其他解释可以参考本表。

表 2.1 匈牙利符号表示法的前缀代码指导说明书

前缀	数据类型(基本类型)
c	字符
b y	字节(无符号字符)

n	短整数和整数(表示一个数)
i	整数
x，y	短整数(通常用于 x 坐标和 y 坐标)
cx，cy	短整数(通常用于表示 x 和 y 的长度： c 表示计数)
b	布尔型(整数)
w	ＵＩＮＴ(无符号整数)和ＷＯＲＤ(无符号字)
l	ＬＯＮＧ(长整数)
dw	ＤＷＯＲＤ(无符号长整数)
fn	函数指针
s	串
sz，str	以 0 字节终止的字符串
lp	３２位的长整数指针
h	编号(常用于表示Ｗｉｎｄｏｗｓ对象)
msg	消息

变量的命名

应用匈牙利符号表示法，变量可用表 2.1 中的前缀代码来表示。另外，当一个变量是由一个或几个子名构成时，每一个子名都要以大写字母开头。下面是几个例子：

```
char *szFileName;//a nulla terminated string
int *lpiDate;//a 32-bit pointer to an int
BOOL bSemaphore;//a boolean value
WORD dwMaxCount;//a 32-bit unsigned WORD
```

尽管我了解一个函数的局部变量没有说明，但是也有个别表示全局变量：

```
int g_iXPos;//a global x-position
int g_iTimer;//a global y-position
char *g_szString;//a global NULL terminated string
```

总的来说，变量以 g_开头，或者有时就只用 g。

函数的命名

函数和变量命名方式相同，但是没有前缀。换句话说，子名的第一个字母要大写。下面是几个例子：

```
int PlotPixel(int ix,int iy,int ic);
void *MemScan(char *szString);
```

而且，下划线是非法的，例如，下面的函数名表示是无效的匈牙利符号表示法：

```
int Get_Pixel(int ix,int iy);
```

类型和常量的命名

所有的类型和常量都是大写字母，但名字中可以允许使用下划线。例如：

```
const LONG NUM_SECTORS = 100;//a C++ style constant
```

```
#define MAX_CELLS 64;//a C style constant
#define POWERUNIT 100;//a C style constant
typedef unsigned char UCHAR;//a user defined type
```

这儿并没有什么不同的地方——非常标准的定义。尽管大多数 Microsoft 程序员不使用下划线，但我还是喜欢用，因为这样能使名字更具有可读性。

C++ 在 C++中，关键字 const 不止一个意思。在前面的代码行中，它用来创建一个常数变量。这和#define 相似，但是它增加了类型信息这个特性，const 不仅仅像#define 一样是一个简单的预处理文本替换，而且更像是一个变量，它允许编译器进行类型检查和替换。

类的命名

类命名的约定可能要麻烦一点。但我也看到有很多人在使用这个约定，并独立地进行补充。不管怎样说，所有 C++的类必须以大写 C 为前缀，类名字的每一个子名的第一个字母都必须大写。下面是几个例子：

```
class CVector
{
  public
  CVector();{ix=iy=yz=imagnitude = 0;}
  CVector(int x,int y,int z){ix=x;iy=y;iz=z;}
  .
  .
  private:
  int ix,iy,iz;//the position of the vector
  int imagnitude;//the magnitude of the vector
};
```

参数的命名

函数的参数命名和标准变量命名的约定相同，但也不总是如此。例如下面例子给出了一个函数定义：

```
UCHAR GetPixel(int x,int y);
```

这种情况下，更准确的匈牙利函数原型是：

```
UCHAR GetPixel(int ix,int iy);
```

但我认为这并没有什么两样。

最后，你甚至可能都看不到这些变量名，而仅仅看到类型，如下所示：

```
UCHAR GetPixel(int, int);
```

当然，这仅仅是原型使用的，真正的函数声明必须带有可赋值的变量名，这一点你已经掌握了。

注意：仅仅会读匈牙利符号表示并不代表你能使用它。实际上，我进行编程工作已经有 20 多年了，我也不准备为谁改变我的编程风格。因此，本书中的代码使用类匈牙利符号表示法的编码风格，这是 Win32 API 造成的，在其他位置将使用我自己的风格，必须注意的是，我使用的变量名的第一个字母没有大写，并且我还使用下划线。

世界上最简单的 Windows 程序

现在读者已经对 Windows 操作系统及其性能和基本设计问题有了一般了解，那就让我们第一个 Windows 程序开始真正的 Windows 编程吧。

以每一种新语言或所学的操作系统来编写一个“世界你好”的程序是一个惯例，让我们来试试，清单 2.1 是标准的基于 DOS 的“世界你好”程序。

程序清单 2.1 基于 DOS 的“世界你好”程序

```
//DEMO2_1.cpp - standard version
#include <stdio.h>
//main entry point for all standard DOS/console programs
void main(void)
{
  printf("\nTHERE CAN RE ONLY ONE!!!\n");
}//end main
```

现在让我们看一看用 Windows 如何编写它。

技巧 顺便说一句，如果读者想编译 DEMO2_1.CPP 的话，就应当用 VC++或 Borland 编译器实际创建一个调用内容的控制应用程序，这是一个类 DOS 的应用程序，只是它是 32 位的，它仅以文本模式运行，但对于检验一个想法和算法是很有用的。

总是从 WinMain()开始

如前面所述,所有的 Windows 程序都以 WinMain()开始,这和简单直观的 DOS 程序都以 Main()开始一样。WinMain()中的内容取决于你。如果你愿意的话，可以创建一个窗口、开始处理事件并在屏幕上画一些东西。另一方面，你可以调用几百个(或者是几千个)Win32 API 函数中的一个。这正是我们将要做的。

我只想在屏幕上的一个信息框中打印一点东西。这正是一个 Win32 API 函数 MessageBox()的功能。清单 2.2 是一个完整的、可编译的 Windows 程序，该程序创建和显示了一个能够到处移动和关闭的信息框。

程序清单 2.2 第一个 Windows 程序

```
//DEMO2_2.CPP - a simple message box
#define WIN32_LEAN_AND_MEAN
#include <windows.h>//the main windows headers
#include <windowsx.h>//a lot of cool macros
//main entry point for all windows programs
int WINAPI WinMain(HINSTANCE hinstance,
HINSTANCE hprevinstance,
LPSTRlpcmdline,
int ncmdshow)
{
//call message box api with NULL for parent window handle
MessageBox(NULL,"THERE CAN BE ONLY ONE!!!",
```

```
"MY FIRST WINDOWS PROGRAM",
MB_OK|MB_ICONEXCLAMATION);
//exit program
return(0);
}//end WinMain
```

要编译该程序，按照下面步骤：

1.创建新的 Win32.EXE 项目并包含 CD-ROM 上 T3DCHAP02\下的 DEMO02_2.CPP。

2.编译和联接程序。

3.运行!

你可能会以为一个基本的 Windows 程序有几百行代码。当你编译和运行程序时，可能会看到如图 2.5 所示的内容。

程序剖析

现在已经有了下完整的 Windows 程序，让我们一行一行地分析程序的内容。首先第一行程序是

```
#define Win32_LEAN_AND_MEAN
```

这个应稍微解释一下。创建 Windows 程序有两种方式——使用 Microsoft 基础类(Microsoft Foundation Classes，MFC)，或者使用软件开发工具包(Software Development Kit,SDK)。MFC 完全基于 C++类，要比以前的游戏编程所需的工具复杂得多，功能和难度也要强大和复杂 10 倍。而 SDK 是一个可管理程序包,可以在一到两周内学会(至少初步学会),并且使用了简单明了的 C 语言。因此，我在本书所使用的工具是 SDK。

Win32_LEAN_AND_MEAN 指示编译器(实际上是逻辑头文件)不包含无关的 MFC 操作，现在我们又离题了，回来继续看程序。

之后，下面的头文件是：

```
#include "windows.h"
#include "windowsx.h"
```

第一个引用“windows.h”实际上包括所有的 Windows 头文件。Windows 有许多这样的头文件，这就有点像包含宏，可以节省许多手工包含显式头文件的时间。

第二个引用“windowsx.h”是一个含有许多重要的宏和常量的头文件,该文件可以简化 Windows 编程。

下面就到最重要的部分——所有 Windows 应用程序的主要入口位置 WinMain()：

```
int WINAPI WinMain(HINSTANCE hinstance,
                   HINSTANCE hprevinstance,
                   LPSTR lpcmdline,
                   int ncmdshow);
```

首先，应当注意到奇怪的 WINAPI 声明符，这相当于 PASCAL 函数声明符，它强制参数从左边向右边传递,而不是像默认的 CDECL 声明符那样参数从右到左转移。但是，PASCAL 调用约定声明已经过时了，WINAPI 代替了该函数。必须使用

WinMain()的 WINAPI 声明符；否则，将向函数返回一个不正确的参数并终止开始程序。

测试参数

下面我们详细看一下每个参数

• hinstance——该参数是一个 Windows 为你的应用程序生成的实例句柄。实例是一个用来跟踪资源的指针或数。本例中，hinstance 就像一个名字或地址一样，用来跟踪你的应用程序。

• hprevinstance——该参数已经不再使用了，但是在 Windows 的旧版本中，它跟踪应用程序以前的实例(换句话说，就是产生当前实例的应用程序实例)。难怪 Microsoft 要去除它，它就像一次长途跋涉——主我们为之头疼。

• lpcmdline——这是一个空终止字符串，和标准 C/C++ main(int argc,char **argv)函数中的命令行参数相似。不同的是，它不是一个单独的像 argc 那样指出命令行的参数。例如，如果你创建一个名字为 TEST.EXE 的 Windows 应用程序，并且使用下面的参数运行：

TEST.EXE one

lpcmdline 将含有下面数据：

lpcmdline = "one two three"

注意，.EXE 的名字本身并不是命令行的一部分。

• ncmdshow——最后一整型参数在运行过程中被传递给应用程序，带有如何打开主应用程序窗口的信息。这样，用户便会拥有一点控制应用程序如何启动的能力。当然，作为一个程序员，如果想忽略它也可以，而想使用它也行。(你将参数传递给 ShowWindow()，我们又超前了！)表 2.2 列出了 ncmdshow 最常用的参数值。

表 2.2　cmdshow 的 Windows 代码

值	功　　能
SW_SHOWNORMAL	激活并显示一个窗口。如果该窗口最小化或最大化的话，Windows 将它恢复到原始尺寸和位置。当第一次显示该窗口时，应用程序将指定该标志。
SW_SHOW	激活一个窗口，并按当前尺寸和位置显示
SW_HIDE	隐藏一个窗口，并激活另外一个窗口
SW_MAXIMIZE	将指定的窗口最大化
SW_MINIMIZE	将指定的窗口最小化
SW_RESTORE	激活并显示一个窗口，如果该窗口最小化或最大化的话，Windows 将它恢复到原始尺寸和位置。当恢复为最小化窗口时,应用程序必须指定该标志。
SW_SHOWMAXIMIZED	激活一个窗口，并以最大化窗口显示
SW_SHOWMINIMIZED	激活一个窗口，并以最小化窗口显示
SW_SHOWMINNOACTIVE	以最小化窗口方式显示一个窗口，激活的窗口依然保持激活的状态

SW_SHOWNA	以当前状态显示一个窗口，激活的窗口依然保持激活的状态
SW_SHOWONACTIVATE	以上一次窗口尺寸和位置来显示窗口，激活的窗口依然保持激活的状态

如表 2.2 所示，ncmdshow 有许多设置(目前许多值都没有意义)。实际上，这些设置大部分都不在 ncmdshow 中传递。可以应用另一个函数 ShowWindow()来使用它们，该函数在一个窗口创建时就开始显示。对此我们在本章后面将进行详细的讨论。

我想说的一点是，Windows 带有大量的你从未使用过的选项和标志等等，就像 VCR 编程选项一样——越多越好，任你使用。Windows 就是按照这种方式设计的。这将使每个人都感到满意，这也意味着它包含了许多选项。实际上，我们在 99%时间内将会使用 SW_SHOW、SW_SHOWNORMAL 和 SW_HIDE，但是你还要了解在 1%的时间内会用到的其他选项。

选择一个信息框

最后让我们讨论一下 WinMain()中调用 MessageBox()的实际机制。MessageBox()是一个 Win32 API 函数，它替我们做某些事，使我们不需自己去做。该函数经常以不同的图标和一个或两个按钮来显示信息。你看，简单的信息显示在 Windows 应用程序中非常普通，有了这样一个函数就节省了程序员半个多小时的时间，而不必每次使用都要编写它。

MessageBox()并没有什么多少功能，但是能够在屏幕上显示一个窗口，提出一个问题，并且等候用户的输入。下面是 MessageBox()的原型：

```
int MessageBox(HWND hwn,//handle of owner window
               LPCTSTR lptext,//address of text in message box
               LPCTSTR lpcaption,//address of title of message box
               UINT utype);//style of message box
```

参数定义如下：

hwnd——这是信息框连续窗口的句柄。目前我们还不能谈及窗口句柄，因此只能认为它是信息框的父窗口。在 DEMO2_2.CPP，我们将它设置为空值 NULL，因此使用 Windows 桌面作为父窗口。

lptext——这是一个包含显示文本的空值终止字符串。

lpcaption——这是一个包含显示文本框标题的空值终止字符串。

utype——这大概是该簇参数中唯一令人激动的参数了，控制信息显示框的各类。

表 2.3 列出了几种 MessageBox()选项(有些删减)。

表 2.3 MessageBox()选项

标志	描述

下列设置控制信息框的一般类型

MB_OK	信息框含有一个按钮：OK，这是默认值
MB_OKCANCEL	信息框含有两个按钮：OK 和 Cancel
MB_RETRYCANCEL	信息框含有两个按钮：Retry 和 Cancel
MB_YESNO	信息框含有两个按钮：Yes 和 No
MB_YESNOCANCEL	信息框含有三个按钮：Yes、No 和 Cancel
MB_ABORTRETRYIGNORE	信息框含有三个按钮：Yes、No 和 Cancel

这一组控制在图标上添加一点“穷人的多媒体”

MB_ICONEXCLAMATION	信息框显示一个惊叹号图标
MB_ICONINFORMATION	信息框显示一个由圆圈中的小写字母 I 构成的图标
MB_ICONQUESTION	信息框显示一个问号图标
MB_ICONSTOP	信息框显示一个终止符图标

该标志组控制默认时高亮的按钮

MB_DEFBUTTONn	其中 n 是一个指示默认按钮的数字(1-4)，从左到右计数

注意：还有其他的高级 OS 级标志，我们没有讨论。如果希望了解更多细节的话，可以通过编译器 Win32 SDK 的在线帮助来查阅。

可以同时使用表 2.3 中的值进行逻辑或运算，来创建一个信息框。一般情况下，只能从每一组中仅使用一个标志来进行或运算。

当然，和所有 Win2 API 函数一样，MessageBox()函数返回一个值业通知编程者所发生的事件。但在这个例子中谁关心这个呢？通常情况下，如果信息框是 yes/no 提问之类的情况的话，就希望知道这个返回值。表 2.4 列出了可能的返回值。

表 2.4 MessageBox()的返回值

值	按钮选择
IDABORT	Abort
IDCANCEL	Cancel

IDIGNORE	Ignore
IDNO	No
IDOK	OK
IDRETRY	Retry
IDYES	Yes

最后，这个表已经毫无遗漏地列出了所有的返回值，正在已经完成了对我们第一个 Windows 程序——单击的逐行分析。

技巧:现在希望你能轻松地对这个程序进行修改,并以不同的方式进行编译，使用不同的编译器选项，例如优化，然后尝试通过调试程序来运行该程序，看看你是否已经领会，做完后，请回到此处。

BOOL MessageBeep(UNIT utype);//运行声音
可以从表 2.5 所示常数中得到不同的声音。

表 2.5 MessageBeep()函数的声音标识符

值	声音
MB_ICONASTERISK	系统星号
MB_ICONEXCLAMATION	系统惊叹号
MB_ICONHAND	系统指针
MB_ICONQUESTION	系统问号
MB_OK	系统默认值
0xFFFFFFFF	使用计算机扬声器的标准嘟嘟声

注意：如果已经安装了 MS_Plus 主题曲的话，你应能得到有意思的结果。

看 Win32 API 多酷啊！可以有上百个函数使用。它们虽然不是世界是最快的函数，但是对于一般的内部管理 I/O 和 GUN 来讲，它们已经很棒了。

让我们稍微花点时间总结一下我们目前所知的有关 Windwos 编程方面的知识。首先，Windows 支持多任务/多线程，因此可以同时运行多个应用程序。我们不必费心就可以做到这一点，我们最关系的是 Windows 支持事件触发。这就意味着我们必须处理事件(在这一点上目前我们还不知如何做)并且做出反应。好，听上去不错。最后所有 Windows 程序都以函数 WinMain()开始，WinMain()函数中的参数要比标准 DOS Main()多得多，但这些参数都属于逻辑和推理的领域。

掌握了上述的内容，就到了编写一个真正的 Windows 应用程序的时候了。

真实的 Windows 应用程序

尽管本书的目标是编写在 Windows 环境下运行的 3D 游戏，但是你并不需要了解更多的 Windows 编程。实际上，你所需要的就是一个基本的 Windows 程序，可以打开一个窗口、处理信息、调用主游戏循环等等。了解了这些，本章中的目标是首先向你展示如何创建简单的 Windows 应用程序,同时为编写类似 32 位 DOS

环境的游戏外壳程序奠定基础。

一个 Windows 程序的关键就是打开一个窗口。一个窗口就是一个显示文本和图形信息的工作区。要创建一个完全实用的 Windows 程序，只要进行下列工作：

1.创建一个 Windows 类。

2.创建一个事件句柄或 WinProc。

3.用 Windows 注册 Windows 类。

4.用前面创建的 Windows 类创建一个窗口。

5.创建一个能够从事件句柄获得或向事件句柄传递 Windows 信息的主事件循环。

让我们详细了解一下每一步的工作。

Windows 类

Windows 实际上是一个面向对象的的操作系统，因此 Windows 中大量的概念和程序都出自 C++。其中一个概念就是 Windows 类。Windows 中的每一个窗口、控件、列表框、对话框和小部件等等实际上都是一个窗口。区别它们的就是定义它们的类。一个 Windows 类就是 Windows 能够操作的一个窗口类型的描述。

有许多预定义的 Windows 类，如按钮、列表框、文件选择器等等。你也可以自己任意创建你的 Windows 类。实际上，你可以为自己编写的每一个应用程序创建至少一个 Windows 类。否则你的程序将非常麻烦。因此你应当在画一个窗口时，考虑一个 Windows 类来作为 Windows 的一个模板，以便于在其中处理信息。

控制 Windows 类信息的数据结构有两个：WNDCLASS 和 WNDCLASSEX。WNDCLASS 是比较古老的一个，可以不久将废弃，因此我们应当使用新的扩展版 WNDLCASSEX。二都结构非常相似，如果有兴趣的话，可以在 Win32 帮助中查阅 WNDCLASS。让我们看一下在 Windows 头文件中定义的 WNDCLASSEX。

```
typedef struct _WNDCLASSEX
{
UINT cbSize;//size of this structure
UINT style;//style flags
WNDPROC lpfnWndProc;//function pointer to handler
int cbClsExtra;//extra class info
int cbWndExtra;//extra window info
HANDLE hInstance;//the instance of the application
HICON hIcon;//the main icon
HCURSOR hCursor;//the cursor for the window
HBRUSH hbrBackground;//the background brush to paint the window
LPCTSTR lpszMenuName;//the name of the menu to attach
LPCTSTR lpszClassName;//the name of the class itself
HICON hIconSm;//the handle of the small icon
} WNDCLASSEX
```

因此你所要做的就是创建一个这样的结构,然后填写所有的字段:

```
WNDCLASSEX winclass;//a blank windows class
```

第一个字段 cbSize 非常重要(但 Petzold 在《Programming Windows 95》忘记了这个内容)，它是 WNDCLASSEX 结构本身的大小。你可能要问，为什么应当知

道该结构的大小？这个问题问得好，原因是如果这个结构作为一个指针被传递的话，接收器首先构件第一个字段，以确定该数据导体最低限度有多大。这有点像提示和帮助信息，以便于其他函数在运行时不必计算该类的大小。因此，我们应当这样做：

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

第二个字段包含描述该窗口一般属性的结构信息标志。有许多这样的标志，因此我们不能全部列它们。只要能够使用它们创建任何类型的窗口就行了。表 2.6 列出了常用的标志。读者可以任意对这些值进行逻辑“或”运算，来派生所希望的窗口类型。

表 2.6 Windows 在的类型标志

标志	说明
CS_HREDRAW	若移动或改变了窗口宽度，则刷新整个窗口
CS_VREDRAW	若移动或改变了窗口高度，则刷新整个窗口
CS_OWNDC	为该类中每个窗口分配一个单值的设备描述表(在本章后面详细描述)
CS_DBLCLKS	当用户双击鼠标时向窗口程序发送一个双击的信息，同时，光标位于属于该类的窗口中
CS_PARENTDC	在母窗口中设定一个子窗口的剪切区，以便于子窗口能够画在母窗口中
CS_SAVEBITS	在一个窗口中保存用户图像，以便于在该窗口被遮住、移动时不必每次刷新屏幕。但是，这样会占用更多的内存，并且比人工同样操作要慢得多
CS_NOCLOSE	禁止系统菜单上的关闭命令

注意：用黑体显示的部分为最常用的标志

表 2.6 包含了大量的标志，即使读者对此尚有疑问，也无关紧要。现在，设定类型标识符，描述如果窗口移动或改变尺寸就进行屏幕刷新，并可以获得一个静态的设备描述表以及处理双击事件的能力。

我们将在第三章“高级 Windows 编程”中详细讨论设备描述表，但基本说来，它被用作窗口中图像着色的数据结构。因此如果你要处理一个图像，就应为感兴趣的特定窗口申请一个设备描述表。如果设定了一个 Windows 灰，它就通过 CS_OMNDC 得到了一个设备描述表，如果你不想每次处理图像时都申请设备描述表，可以将它保存一段时间。上面说的对你有帮助还是使你更糊涂？Windows 就是这样——你知道得越多，问题就越多。旭了！下面说一下如何设定类型字段：

```
winclass.style = CS_VERDRAW | CS_HREDRAW | CS_OWNDC | CS_DBLCLICKS;
```

WNDCLASSEX 结构的下一个字段 lpfnWndProc 是一个指向事件句柄的函数指针。基本上这里所设定的都是该类的回调函数。回调函数在 Windows 编程中经常

使用，工作原理如下：当有事件发生时，Windows 通过调用一个你已经提供的回调函数来通知你，这省去你盲目查询的麻烦。随后在回调函数中，再进行所需的操作。

这个过程就是基本的 Windows 事件循环和事件句柄的操作过程。向 Windows 类申请一个回调函数(当然需要使用特定的原型)。当一个事件发生时，Windows 按如图 2.6 所示的那样替你调用它。关于该项内容我们将在下面部分进行更详细的介绍。但是现在，读者只要将其设定到你将编写的事件函数中去：

```
winclass.lpfnWndProc = WinProc;//this is our function
```

----------(图 2.6　windows 事件句柄回调函数的操作流程　略)

C++ 函数指针有点像 C++中的虚函数。如果你对它们不熟悉的话，在这里讲一上，假设有两个函数用以操作两个数：

```
int Add(int op1,int op2) {return(op1+op2);}
int Sub(int op1,int op2) {return(op1-op2);}
```

要想用同一调用来调用两个函数，可以用一个函数指针来实现，如下：

```
//define a function pointer that takes two int and returns an int
int (Math*)(int,int);
```

然后可以如下配置函数指针：

```
Math = Add;
int result = Math(1,2);//this reslly calls Add(1,2)
//result will be 3
Math = Sub;
int result = Math(1,2);//this really calls Sub(1,2)
//result will be -2
```

看，不错吧。

下面两个字段，cbClsExtra 和 cbWndExtra 原是为指示 Windows 将附加的运行时间信息保存到 Windows 类某些单元中而设计的。但是绝大多数人使用这些字段并简单地将其值设为 0，如下所示：

```
winclass.cbClsExtra = 0;//extra class info space
winclass.cbWndExtra = 0;//extra window info space
```

下一个是 hInstance 字段。这是一个简单的、在启动时传递给 WinMain()函数的句柄实例，因此只需简单地从 WinMain()中复制即可：

```
winclass.hInstance = hinstance;//assign the application instance
```

剩下的字段和 Windows 类的图像方面有关，在讨论它们之前，先花一点时间回顾一下句柄。

在 Windows 程序和类型中将一再看到句柄：位图句柄、光标句柄、任意事情的句柄。请记住，句柄只是一个基于内部 Windows 类型的标识符。其实它们都是整数。但是 Microsoft 可能改变这一点，因此安全使用 Microsoft 类型是个好主意。总之，你将会看到越来越多的“[...]句柄”，请记住，有前缀 h 的任何类型通常都是一个句柄。好，回到原来的地方继续吧。

下一个字段是设定表示应用程序的图标的类型。你完全可以装载一下你自己定制的图标，但现在你使用系统图标，需要为它设置一个句柄。要为一个常用的系统图标检索一个句柄，可以使用 LoadIcon()函数：

winclass.hIcon = loadIcon(NULL,IDI_APPLICATION);

这行代码装载一个标准的应用程序图标——虽然烦人，但是简单。如果对LoadIcon()函数有兴趣的话，请看下面的它的原型，表 2.7 给出了几个图标选项：

HICON LoadIcon(HINSTANCE hInstance,//handle of application instance
LPCTSTR lpIconName;//icon-name string or icon resource identifier

hInstance 是一个从应用程序装载图标资源的实例(后面将详细讨论)。现在将它设置为 NULL 来装载一个标准的图标。LpIconName 是包含被装载图标资源名称的 NULL 终止字符串。当 hInstance 为 NULL 时，IpIconName 的值如表 2.7 所示。

LoadIcon()的图标标识符

值	说明
IDI_APPLICATION	默认应用程序图标
IDI_ASTERISK	星号
IDI_EXCLAMATION	惊叹号
IDI_HAND	手形图标
IDI_QUESTION	问号
IDI_WINLOGO	Windows 徽标

好，现在我们已经介绍了一半的字段了。做个深呼吸休息一会，让我们进行下一个字段 hCursor 的介绍。和 hIcon 相似，它也是一个图像对象句柄。不同的是，hCursor 是一个指针进入到窗口的用户区才显示的光标句柄。使用LoadCursor()函数可以得到资源或预定义的系统光标。我们将在后面讨论资源，简单而言资源就是像位图、光标、图标、声音等一样的数据段，它被编译到应用程序中并可以在运行时访问。Windows 类的光标设定如下所示：

winclass.hCursor = LoadCursor(NULL,IDC_ARROW);

下面是 LoadCursor()函数的原型(表 2.8 列出了不同的系统光标标识符)：

HCURSOR lpCursor(HINSTANCE hInstance,//handle of application instance
LPCTSTR lpCursorName);//icon_name string or icon resource identifier

hInstance 是你的.EXE 的应用程序实例。该.EXE 应用程序包含订制光标名称来源的资源。但现在读者还不能使用该功能，仅将默认的系统光标值设定为 NULL。

lpCursorName 标识了资源名字符串或资源句柄(我们一般不使用)，或者是一个常数，以标识如表 2.8 中所示的系统默认值。

表 2.8 LoadCursor()的值

值	说明

IDC_ARROW	标准箭头
IDC_APPSTARTING	标准箭头和小沙漏标
IDC_CROSS	横标线
IDC_IBEAM	文本 I 型标
IDC_NO	带正斜线的圆圈
IDC_SIZEALL	四向箭头
IDC_SIZENESW	指向东北-西南方向的双向箭头
IDC_SIZENS	指向南北方向的双向箭头
IDC_SIZENWSE	指向东南-西北方向的双向箭头
IDC_SIZEWE	指向东西方向的双向箭头
IDC_UPARROW	垂直方向的箭头
IDC_WAIT	沙漏

现在我们就要解放了，因为我们几乎已经全部介绍完了——剩下的字段更有意义。让我们看一看 hbrBackground。

无论在什么时候绘制或刷新一个窗口，Windows 都至少将以用户预定义的颜色或 Windows 内部设置的画笔颜色填充该窗口的背景。因此，hbrBrackground 是一个用于窗口刷新的画笔句柄。画笔、笔、色彩和图形都时 GDI(图形设备接口)的一部分，我们将在下一章中详细讨论。现在，介绍一下如何申请一个基本的系统画笔来填充窗口。该项功能由 GetStockObject()来实现，如下面程序所示：

```
winclass.hbrBackground = GetStockObject(WHITE_BRUSH);
```

GetStockObject()是一个通用函数，用于获得 Windows 系统画笔、笔、调色板或字体的一个句柄。GetStockObject()只有一个参数，用来指示装载哪一项资源。表 2.9 仅列出了画笔和笔的可能库存对象。

表 2.9 GetStockObject()的库存对象标识符

值	说明
BLACK_BRUSH	黑色画笔
WHITE_BRUSH	白色画笔
GRAY_BRUSH	灰色画笔
LTGRAY_BRUSH	淡灰色画笔
DKGRAY_BRUSH	深灰色画笔
HOLLOW_BRUSH	空心画笔
NULL_BRUSH	无效(NULL)画笔
BLACK_PEN	黑色笔
WHITE_PEN	白色笔
NULL_PEN	无效(NULL)笔

WNDCLASS 结构中的下一个字段是 lpszMenuName。它是菜单资源名称的空终止 ASCII 字符串，用于加载和选用窗口。其工作原理将在第三章“高级 Windows 编程”中讨论。现在我们只需要将值设为 NULL。

```
winclass.lpszmenuNume = NULL://the name of the menu to attach
```

如我刚提及的那样，每个 Windows 类代表你的应用程序所创建的不同窗口类型。在某种程序上，类与模板相似，Windows 需要一些途径来跟踪和识别它们。因此，下一个字段 lpszClassName，就用于该目的。该字段被赋以包含相关类的文本标示符的空终止字符串。我个人喜欢用诸如“WINCLASS1”、“WINCLASS2”等标示符。读者以自己喜好而定，以简单明了为原则，如下所示：

```
Winclass.lpszClassName="WINCLASS1"//the name of the class itself
```

这样赋值以后，你可以使用它的名字来引用这个新的 Windows 类——很酷，是吗？

最后就是小应用程序图表。这是 Windows 类 WINCLASSEX 中新增加的功能，在老版本 WNDCLASS 中没有。首先，它是指向你的窗口标题栏和 Windows 桌面任务样的句柄。你经常需要装载一个自定义资源，但是现在只要通过 LoadIcon() 使用一个标准的 Windows 图标即可实现：

```
winclass.hIconSm = LoadIcon(NULL,IDI_APPLICATION);//小图标句柄
```

下面让我们迅速回顾一下整个类的定义：

```
WNDCLASSEX winclass;//this will hole the class we create
//first fill in the window class structure
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL,IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL,IDC_ARROW);
winclass.hbrBacground = GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = "WINCLASS";
winclass.hIconSm = LoadIcon(NULL,IDI_APPLICATION);
```

当然，如果想节省一些打字时间的话，可以像下面这样简单地初始化该结构：

```
WNDCLASSEX winclass = {
    winclass.cbSize = sizeof(WNDCLASSEX);
    CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW,
    WindowProc,
    0,
    0,
    hinstance,
    LoadIcon(NULL,IDI_APPLICATION),
    LoadCursor(NULL,IDC_ARROW),
    GetStockObject(BLACK_BRUSH),
```

```
    NULL,
    "WINCLASS1",
    LoadIcon(NULL,IDI_APPLICATION)};
```

这样就省去了许多输入！

注册 Windows 类

现在 Windows 类已经定义并且存放在 winclass 中，必须将新的类通知 Windows。该功能通过 RegisterClassEx()函数，使用一个指向新类定义的指针来完成，如下所示：

```
RegisterClassEx(&winclass);
```

警告：注意我并没有使用我们例子中的“WINCLASS1”的类名，对于 RegisterClassEx()来讲，必须使用保存该类的实际结构，因为在该类调用 RegisterClassEx()函数之前，Windows 并不知道该类的存在，明白了吧？

此外还有一个旧版本的 RegisterClass()函数，用于注册基于旧结构 WINCLASS 基础上的类。该类一旦注册，我们就可以任意创建它的窗口。请看下面如何进行这个工作，然后再详细看一下事件句柄和主事件循环，了解使一个 Windows 应用程序运行还要做哪些工作。

创建窗口

要创建一个窗口(劳动者一个类窗口的对象)，使用 CreateWindow()或 CreateWindowEx()函数。后者是更新一点的版本，支持附加类型参数，我们就使用它。该函数是创建 Windows 类的函数，我们要多花一点时间来逐行分析。在创建一个窗口时。必须为这个 Windows 类提供一个正文名——我们现在就使用“WINCLASS1”命名。这是识别该 Windows 类并区别于其他类以及内嵌的诸如按钮、文本框等类型的标示。

下面是 CreateWindowEx()函数的原型：

```
HWND CreateWindowEx(
  DWORD dwExStyle,//extended window style
  LPCTSTR lpClassName,//pointer to registered class name
  LPCTSTR lpWindowName,//pointer to window name
  DWORD dwStyle,//window style
  int x,//horizontal position of window
  int y,//vertical position of window
  int nWidth,//window width
  int nHeight,//widow height
  HWND hWndParent,//handle to parent or owner window
  HMENU hMenu,//handle to menu,or child-window identifier
  HINSTANCE hInstance,//handle to application instance
  LPVOID lpParam);//pointer to window-creation data
```

如果该函数执行正确的话，将返回五个指向新建窗口的句柄；否则就返回空值 NULL。上述大多数参数是不需要加以说明的，现在让我们浏览一下：

- **dwExStyle**——该扩展类型标志具有高级特征，大多数情况下，可以设为 NULL。如果读者对其取值感兴趣的话，可以查阅 Win32 SDK 帮助，上面有详细的

有关该标识符取值的说明。WS_EX_TOPMOST 是我唯一使用过的一个值，该功能使窗口一直保持在上部。

• lpClassName——这是你所创建的窗口的基础类名——例如“WINCLASS1”。

• lpWindowName——这是包含窗口标题的空终止文本字符串——例如“我的第一个窗口”。

• dwStyle——这是一个说明窗口外观和行为的通用窗口标志——非常重要！表 2.10 列出了一些最常用的值。当然，可以任意组合使用这些值来得到希望的各种特征。

• x，y——这是该窗口左上角位置的像素坐标。如果你无所谓，可使用 CW_USEDEFALT，这将由 Windows 来决定。

• nWidth，nHeight——这是以像素表示的窗口宽度和高度。如果你无所谓，可使用 CW_USEDEFAULT，这将由 Window 来决定。

• hWndParent——假如父窗口存在，这是指向父窗口的句柄。如果没有父窗口，桌面就是父窗口。

• hMenu——这是指向附属于该窗口菜单的句柄。下一章中将详细介绍，现在将其赋值 NULL。

• hInstance——这是应用程序实例，这里从 WinMain()中使用实例。

• lpParam——高级特征，设置为 NULL。

表 2.10 列出了各种窗口标志设置。

表 2.10 dwStyle 的通用类型值

类型	所创建的内容
WS_POPUP	弹出式窗口
WS_OVERLAPPED	带有标题栏和边界的重叠式窗口，类似 WS_TILED 类型
WS_OVERLAPPEDWINDOW	具有 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME、WS_MAXIMIZEBOX 和 WS_MINIMIZEBOX 样式的重叠式窗口
WS_VISIBLE	开始就可见的窗口
WS_SYSMENU	标题栏上有窗口菜单的窗口
WS_BORDER	有细线边界的窗口
WS_CAPTION	有标题栏的窗口(包括 WS_BORDER 样式)
WS_ICONIC	开始就最小化的窗口，类似 WS_MINIMIZE 样式
WS_MAXIMIZE	开始就最大化的窗口
WS_MAXIMIZEBOX	具有最大化按钮的窗口。不能和 WS_EX_CONGTEXTHELP 样式合并。WS_SYSMENU 也必须指定
MS_MINIMIZE	开始就最小化的窗口，类似 WS_ICONIC 样式
WS_MINIMIZEOBX	具有最小化按钮的窗口。不能和 WS_EX_CONGTEXITHELP 样式合并。WS_SYSMENU 也必须指定

WS_POPUPWINDOW	带有 WS_BORDER、WS_POPUP 和 WS_SYSMENU 类型的弹出式窗口
WS_SIZEBOX	一个窗口边界可以变化，和 WS_THICKFRAME 类型相同
WS_HSCROLL	带有水平滚动条的窗口
WS_VSCROLL	带有垂直滚动条的窗口

注意：用黑体显示的是经常使用的值。

下面是使用标准控件在(0,0)位置创建一个大小为 400×400 像素的、简单的重叠式窗口。

```
HWND hwnd;//window handle
//create the window,bail if problem
if(!(hwnd = CreateWindowEx(NULL,//extended style
  " WINCLASS";//class
  " Your Basic Window",//title
  WS_OVERLAPPEDWINDOW|WS_VISIBLE;
  0,0 //initial x,y
  400,400 /initital width,height
  NULL,//handle to parent
  NULL,//handle to menu
  hinstance,//instance of this application
  NULL)))//extra creation parms
return(0);
```

一旦创建了该窗口，它可能是可见或不可见的。但是，在这个例子中，我们增加了自动显示的类型标识符 WS_VISIBLE。如果没有添加该标识符，则调用下面的函数来人工显示该窗口：

```
//this shows the window
ShowWindow(hwnd,ncmdshow);
```

记住 WinMain()中的 ncmdshow 参数吗？这就是使用它的方便之处。尽管我们使用 WS_VISIBLE 覆盖了 ncmdshow 参数，但还是应将其作为一个参数传递给 ShowWindow()。下面让 Windows 更新窗口的内容，并且产生了一个 WM_PAINT 信息，这通过调用函数 UpdateWindow()来完成：

```
//this sends a WM_PAINT message to window and makes
//sure the contents are refreshed
UpdateWindow();
```

事件处理程序

我并不了解你的情况，但注意我现在正使你掌握 Windows 的核心。它有如一本神秘小说。请记住，我所说的事件处理程序就是当事件发生时 Windows 从主事件循环调用的回调函数。回顾一下图 2.6，刷新一下你对通用数据流的印象。

事件处理器由读者自己编写，它能够处理你所关心　的所有事件，蓁的工作就交给 windows 处理。当然，请记住，你的应用程序所能处理的事件和消息越多，它的功能越强。

在编写程序之前，让我们讨论一下事件处理器的一些细节，即事件能做什么，工作机理如何。首先，对于创建的一个 Windows 类，都有一个独立的事件处理器，我指的是 Windows' procedure，从现在开始简称 WinProc。当收到用户或 Windows 发送的消息并放在主事件序列中时，WindProc 就接收到主事件循环发送的消息。这简单是一个智力绕口令，我换个方式来说明……

当用户和 Windows 运行任务时，你的窗口和/或其他应用程序窗口产生事件和消息。所有消息都进入一个队列，而你窗口的消息发送到你的窗口专用队列中。然后主事件循环检索这些消息，并且将它们发送到你的窗口的 WinProc 中来处理。

这几乎有上而个可能的消息和变量，因此，我们就不全部分析了。值得调研员幸的是，你只需处理很少的消息和变量，就可以启动并运行 Windows 应用程序。

简单地说，主事件循环将消息和事件反馈到 WinProc，WinProc 对它们进行处理。因此不仅你要关注 WinProc，主事件循环同样也要关心 WinPorc。现在我们简单地了解一下 WinPorc，现 WinProc 只接收消息。

正在来看一下 WinProc 的工作机掉，让我们看一下它的原型：

```
LRESULT CALLBACKWindowProc{
    WHND hwnd,//window handle of sender
    UINT msg,//the message id
    WPARAM wparam,//further defines message
    LPARAM lparam);//further defines message
```

当然，这仅仅是回调函数的原型。只要将函数地址作为一个函数指针传递给 winclass.lpfnWndProc，就可以调用该函数的任何信息，如下如示：

```
winclass.lpfnWndProc = WindowProc;
```

参数的含义是不言自明的：

• hwnd——这是一个 Windows 句柄，只有当你使用同一个 Windows 打开多个窗口时它才用到。这种情况下，hwnd 是表明消息来自哪个窗口唯一途径。图 2.7 表示了这种情况。

• msg——这是一个实际的 WinProc 处理的消息标识符。这个标识符可以是众多主要消息中的一个。

• Wparam 和 lparam——进一步限定或细发送到 msg 参数中的信息。

最后，我们感兴趣的是返回类型 LRESULT 和声明说明符 CALLBAK。这些关键字都是必需的，不能忽略它们。

因此大多数人所要做的就是使用 switch()来处理 msg 所表示的消息，然后为每一种情况编写代码。在 msg 基础上，你可以知道是否需要进一步求 wparam 和/或 lparam 的值。很酷吗？因此让我们看一下由 WinProc 传递过来的所有可能的消息，然后看一下 WinProc 的工作机理。表 2.11 简单列出了一些基本的消息说明符。

类型	所创建的内容
WN_ACTIVATE	当窗口被激活或者成为一个焦点时传递
WM_CLOSE	当窗口关闭时传递

WM_CREATE	当窗口第一次创建时传递
WM_DESTROY	当窗口可能要被破坏时传递
WM_MOVE	当窗口移动时传递
WM_MOUSEMOVE	当移动鼠标时传递
WM_KEYUP	当松开一个键时传递
WM_KEYDOWN	当按钮一下键时传递
WM_TIMER	当发生定时程序事件时传递
WM_USER	允许传递消息
WM_PAINT	当一个窗口需重画时传递
WM_QUIT	当 Windows 应用程序最后结束时传递
WM_SIZE	当一个窗口改变大小时传递

要认真看表 2.11，了解所有消息的功能。在应用程序运行时将有一个或多个上述消息传递到 WinProc。消息说明符本身在 msg 中，而其他停止都存储在 wparam 和 lparam 中。因此参考在线 Win32SDK 帮助来了解某个消息的参数所代表的意思是个不错的方法。

幸好我们现在只对下面三个消息感兴趣：

• WM_CREATE——当窗口第一次创建时传递该消息，以便你进行启动、初始化或资源配置工作。

• WM_PAINT——当一个窗口内容需要重画时传递该消息。这可能有许多原因：用户移动窗口或改变其尺寸、弹出其他应用程序而遮挡了你的窗口等。

• WM_DESTROY——当窗口可能要被破坏时该消息传递到你的窗口。通常　这是由于用户单击该窗口的关闭按钮，或者是从该窗口的系统菜单中关闭该窗口造成的。无论上述哪一种方式，都应用释放所有的资源，并且通过发送一个 WM_QUIT 消息来通知 Windows 完全终止应用程序。后面还将详细介绍。

OK！让我们看一看 WinPorc 处理这些消息的整个过程。

```
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam);
{//this if the main message haneler of the system
  PAINTSTRUCT ps;//Used in WM_PAINT

  HDC hdc;//handle to a device context
  //what is the message
  switch(msg)
  {
    case WM_CREATE;
    {
      //do initialization stuff here
      //return success
      return(0)
    }break;
```

```
    case WM_PAINT;
    {
      //simply validate the window
      hdc = BeginPaint(hwnd,&ps);
      //you woudle do all your painting here
      EndPaint(hwnd,&ps);
      //return success
      return(0)
    }break;
    case WM_DESTROY;
    {
      //Kill the application,this sends a WM_QUIT message
      PostQuitMessage(0);
      //return success
      return(0)
    }break;
    default:break;
  }//end switch
  //process any message that we didn't take care of
  return (DefWindowProc(hwnd,msg,wparam,lparam));
}//end WinProc
```

由上面可以看到，函数的大部分是由空白区构成——这真是件好事件。让我们就以 WM_CREATE 处理程序开始吧。该函数所作的一切就是 return(0)。这就是通知 Windows 由编程人员自己处理该函数，因此就无需更多的操作。当然，也可以在 WM_CREATE 消息中进行全部的初始化工作，但那是你的事了。

下一个消息 WM_PAINT 非常重要。该消息在窗口需要重画时被发送。一般来说这表示你应当进行重画工作。对于 DirectX 游戏来说，这并不是件什么大事，因为你将以 30 到 60 帧/秒的频率来重画屏幕。但是对于标准 Windows 应用程序来说，它就是件大事了。我将在后面章节中更详细地介绍 WM_PAINT，目前的功能就是通知 Windows，你要重画该窗口了，因此就停止发送 WM_PAINT 消息。

要完成该功能，你必须激活该窗口的窗户区。有许多方法可以做到，但调用函数 BeginPaint()和 EndPaint()最简单。这一对调用将激活窗口，并使用原先存储在 Windows 类中的变量 hbrbackground 的背景刷来填充背景。下面是程序代码。

```
//begin painting
hdc = BdginPaint(hwnd,&ps);
//you would do all your painting here
EndPaint(hwnd,&ps);
```

下面要提醒几件事情。第一，请注意，每次调用的第一个参数是窗口句柄 hwnd。这是一个非常必要的参数，因为 BeginPaint——EndPaint 函数能够在任何应用程序窗口中绘制，因此该窗口句柄指示了要重画哪个窗口。第二个参数是包含必须重画矩形区域的 PAINTSTRUCT 结构的地址。下面是 PAINTSTRUCT 结构：

```
typedef struct tagPAINTSTRUCT
```

```
{
HDC hdc;
BOOL fErase;
RECT rcPaint;
BOOL fRestore;
BOOL fIncUpdate;
BYTE rgbReserved[32];
} PAINTSTRUCT;
```

实际上不需要关心这个函数，当我们讨论图形设备接口时会再讨论这个函数。其中最重要的字段就是rcPaint。图2.8表示了这个字段的内容。注意Windows一直尽可能地谋略作最少的工作，因此当一个窗口内容破坏之后，Windows至少会告诉你要恢复该内容并能够重画的最小的矩形。如果你对矩形结构感兴趣的话，会发现只有矩形的四个角是最重要的，如下所示：

```
typedef struct tagRECT
{
LONG left;//left x-edge of rect
LONG top;//top yoedge of rect
LONG right;//right x-edge of rect
LONG bottom;//bottom y-edge of rect
}//RECT;
```

调用BeginPaint()函数应注意的最后一件事情是，它返回一个指向图形环境或hdc的句柄：

```
HDC hdc;//handle to graphics context
hdc = BeginPaint(hwnd,&ps);
```

图2.8 仅重新绘制无效区

图形环境就是描述视频系统和正在绘制表面的数据结构。奇妙的是，如果你需要绘制图形的话，只要获得一个指向图形环境的句柄即可。这便是关于WM_PAINT消息内容。

WM_DESTROY消息实际上非常有意思。WM_DESTROY在用户关闭窗口时被发送。当然仅仅是关闭窗口，而不是关闭应用程序。应用程序继续运行，但是没有窗口。对此要进行一些处理。大多数情况下，当用户关闭主要窗口时，也就意味着要关闭该应用程序。因此，你必须通过发送一个消息来通知系统。该消息就是WM_QUIT。因为该消息经常使用，所以有一个函数PostQuitMessage()来替你完成发送工作。

在WM_DESTROY处理程序中你所要做的就是清除一切，然后调用PostQuitMessage(0)通知Windows终止应用程序。接着将WM_QUIT置于消息队列，这样在某一个时候终止主事件循环。

在我们所分析的WinProc句柄中还有细节应当了解。首先，你肯定注意到了每个处理程序体后面的return(0)。它有两个目的：退出WinPorc以及通知Windows你已处理的信息。第二个重要的细节是默认消息处理程序DefaultWindowProc()。该函数是一个传递Windows默认处理消息的传递函数。因此，如果不处理该消息的话，可通过如下所示的调用来结束你的所有事件处理函数：

```
//porcess any message that we didn't take card of
return (DefWindowProc(hwnd,msg,wparam,lparam));
```

我认为代码本身过多并且过于麻烦。然而，一旦你有了一个基本 Windows 应用程序架构的话，你只要将它复制并在其中添加你自己的代码就行了。正如我所说的那样，我的主要目标是帮助你创建一个可以使用的类 DOS32 的游戏操作台，并且几乎忘记了任何正在运行的 Windows 工作。让我们转到下一部分——主事件循环。

主事件循环

最难的一部分终于结束了。我正要脱口而出：主事件循环太简单了。下面讨论一下：

```
//enter main event loop
while(GetMessage(&msg,NULL,0,0))
{
//translate any accelerator keys
TranslateMessage(&msg);
//send the message to the window proc
DispatchMessage(&msg);
}//end while
```

这是什么？OK！让我们来研讨一下。只要 GetMessage()返回一个非零值，主程序 while()就开始执行。GetMessage()是主事件循环的关键代码，其唯一的用途就是从事件队列中获得消息，并进行处理。你会注意到 GetMessage()有四个参数。第一个参数对我们非常重要，而其余的参数都可以设置为 NULL 或 0。下面列出其原型，以供参考：

```
BOOL GetMessage{
  LPMSG lpMsg,//address of structure with message
  HWND hWnd,//handle of window
  UINT wMsgFileterMin,//first message
  UINT wMsgFileterMax);//last message
```

msg 参数是 Windows 放置下一个消息的存储器。但是和 WinProc()的 msg 参数不同的是，该 msg 是一修复杂的数据结构，而不仅仅是一个整数。当一个消息传递到 WinProc 时，它就被处理并分解为各个组元。MSG 的结构如下所示：

```
typedef struct tagMSG
{
HWND hwnd,//window where message occurred
UINT message;//message id itself
WAPARAM wParam;//sub qualifies message
LPARAM lParam;//sub qualifies message
DWORD time;//time if message event
POINT pt;//position of mouse
}MSG;
```

看出点眉目来了，是吗？注意所有向 WinPorc()传递的参数都包含在该结构中，还包括其他参数，如事件发生时的时间和鼠标的位置。

GetMessage()从时间序列中获得下一个消息，然后下一个被调用的函数就是TranslateMessage()。TranslateMessage()是一个虚拟加速键转换器——换句话说就是输入工具。现在只是调用它，不必管其功能。最后一个函数DispatchMessage()指出所有操作发生的位置。当消息被 GetMessage 获得以后，由函数 TranslateMessae()稍加处理和转换，通过函数 DispatchMessage()调用 WinProc 进行进一步的处理。

DispatchMessage()调用 WinProc，并从最初的 MSG 结构中传递适当的参数。图 2.9 表示了整个处理过程的最后部分。

这样，你就成了 Windows 专家了！如果你已经理解了上面刚刚讨论过的概念以及事件循环、事件处理程序等等的重要性，那你至少已经掌握了 90%的内容了。剩下的就是一些细节问题了。

程序清单 2.3 是一个完整的 Windows 程序，内容是创建一个窗口。并等候关闭。

程序清单 2.3 一个基本的 Windows 程序

```
--------------------
// DEMO2_3.CPP - A complete windows program

// INCLUDES ///////////////////////////////////////////
#define WIN32_LEAN_AND_MEAN // just say no to MFC

#include <windows.h> // include all the windows headers
#include <windowsx.h> // include useful macros
#include <stdio.h>
#include <math.h>

// DEFINES ////////////////////////////////////////////

// defines for windows
#define WINDOW_CLASS_NAME "WINCLASS1"

// GLOBALS ////////////////////////////////////////////


// FUNCTIONS //////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
UINT msg,
WPARAM wparam,
LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT ps; // used in WM_PAINT
HDC hdc; // handle to a device context

// what is the message
switch(msg)
```

```
{
case WM_CREATE:
{
// do initialization stuff here

// return success
return(0);
} break;

case WM_PAINT:
{
// simply validate the window
hdc = BeginPaint(hwnd,&ps);
// you would do all your painting here
EndPaint(hwnd,&ps);

// return success
return(0);
} break;

case WM_DESTROY:
{
// kill the application, this sends a WM_QUIT message
PostQuitMessage(0);

// return success
return(0);
} break;

default:break;

} // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////
int WINAPI WinMain( HINSTANCE hinstance,
HINSTANCE hprevinstance,
LPSTR lpcmdline,
int ncmdshow)
{
```

```
WNDCLASSEX winclass; // this will hold the class we create
HWND hwnd; // generic window handle
MSG msg; // generic message

// first fill in the window class stucture
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS | CS_OWNDC |
CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName = NULL;
winclass.lpszClassName = WINDOW_CLASS_NAME;
winclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

// register the window class
if (!RegisterClassEx(&winclass))
return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL, // extended style
WINDOW_CLASS_NAME, // class
"Your Basic Window", // title
WS_OVERLAPPEDWINDOW | WS_VISIBLE,
0,0, // initial x,y
400,400, // initial width, height
NULL, // handle to parent
NULL, // handle to menu
hinstance,// instance of this application
NULL))) // extra creation parms
return(0);

// enter main event loop
while(GetMessage(&msg,NULL,0,0))
{
// translate any accelerator keys
TranslateMessage(&msg);

// send the message to the window proc
DispatchMessage(&msg);
} // end while
```

```
// return to Windows like this
return(msg.wParam);

} // end WinMain

//////////////////////////////////////////////////////////
```

要编译 DEMO2.3.CPP，只需创建一个 Win32 环境下的.EXE 应用程序，并且将 DEMO2_3.CPP 添加到项目中即可。假如你喜欢的话，可以直接从 CD-ROM 上运行预先编译好的程序 DEMO2_3.EXE。图 2.10 显示了运行中的该程序。

图 2.10 运行中的 DEMO2_3.EXE

在进行下一部分内容之前，我还有事情要说。首先，如果你认真阅读了事件循环的话，会发现它看上去并不是个实时程序。也就是说，当程序在等待通过 GetMessage()传递的消息的同进，主事件循环基本是锁定 的。这的确是真的：你必须以各种方式来避免这种现象，因为你需要连续地运行你的游戏处理过程，并且在 Windows 事件出现时处理这些事件。

产生一个实时事件循环

这种实时的无等侯的事件循环很容易实现.你所需要的就是一种测试在消息序列中是否有消息的方法。如果不，你就处理它；否则，继续处理其他的游戏逻辑并重复进行。运行的测试函数是 PeekMessage()。其原形几乎和 GetMessage()相同，如下所示：

```
BOOL PeekMessage{
  LPMSG lpMag,//pointer to structeure for message
  HWND hWnd,//handle to window
  UINT wMsgFilterMin,//first message
  UINT wMsgFilterMax,//last message
  UINT wRemoveMsg);//removal flags
```

如果有可用消息的话返回值非零。

区别在于最后一个参数，它控制如何从消息序列中检索消息。对于 wRemoveMsg,有效的标志有：

- PM_NOREMOVE——PeekMessage()处理之后，消息没有从序列中去除。
- PM_REMOVE——PeekMessage()处理之后，消息已经从序列中去除。

如果将这两种情况考虑进去的话，你可以做出两个选择：如果有消息的话，就使用 PeekMessage()和 PM_NOREMOVE，调用 GetMessate()；另一种选择是：使用 PM_REMOVE。如果有消息则使用 PeekMessage()函数本身来检索消息。一般使用后一种情况。下面是核心逻辑的代码，我们在主事件循环中消作改动以体现这一新技术：

```
while(TRUE)
{
  //test if there is a message in queue,if so get it
  if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)
```

```
    {
      //test if this a quit
      if(msg.message == WM_QUIT)
        break;
      //translate any accelerator keys
      TranslateMessage(&msg);
      //send the message to the window proc
      DispatchMessage(&msg);
    }//end if
    //main game processing goes here
    Game_Main();
  }//end while
```

我已经将程序中的重要部分用黑体显示。黑体的第一部分内容是：

```
if(msg.message == WM_QUIT) break;
```

下面是如何测试从无限循环体 while(true)中退出。请记住，当在 WinProc 中处理 WM_DESTROY 消息时，你的工作就是通过调用 PostQuitMessage()函数来传递 WM_QUIT 消息。WM_QUIT 就在事件序列中慢慢地移动，你可以检测到它，所以可以跳出主循环。

用黑体显示的程序最后一部分指出调用主游戏程序代码循环的位置。但是请不要忘记，在运行一幅动画或游戏逻辑之后，调用 Game_Main()或者调用任意程序必须返回。否则，Windows 主事件循环将不处理消息。

这种新型的实时结构的例子非常适合于游戏逻辑处理程序，请看源程序 DEMO2_4.CPP 以及 CD-ROM 上相关的 DEMO2_4.EXE。这种结构实际上是本书剩下部分的模型。

打开多个窗口

在完成本章内容之前，我想讨论一个你可能非常关心的更重要的话题——如果打开多个窗口。实际上，这是小事一桩，其实你已经知道如何打开多个窗口。你所需要做的就是多次调用函数 CreateWindowEx()来创建这些窗口，事实也的确如此。但是，对此还有一些需要注意的问题。

首先，请记住当你创建一个窗口时，是建立在 Windows 类的基础之上的。在该类中定义了 WinProc 或者整个类的事件处理程序。这是非常重要的细节，因此应当注意。你可以使用同一个类来创建多个窗口，这些窗口的所有消息都要传递到同一个 WinProc 中，正如由 WINCLASSEX 结构中定义 lpfnWndProc 字段指向的事件处理程序一样。图 2.11 表示了这种情况下的消息流。

------图

这样可能达到你的愿望，也可能达不到。如果想应用不同的 WinProc 打开各个窗口的话，你必须创建多个 Windows 类，并且使用每一个类创建各个窗口。这亲每一个类的窗口就指向各自的 WinProc 并同其传递消息。图 2.12 表示了这种情况。

图 2.12 多个窗口的多个 Windows 类
了解了这些内容后，下面是基于同一个类来创建多个窗口的程序代码：

```
//create the first window
if(!(hwnd = CreateWindowEx(NULL,//extended style
    WINDOW_CLASS_NAME,//class
    "Window 1 Based on WINCLASS1",//title
    WS_OVERLAPPEDWINDOW|WS_VISIBLE,
    0,0,//initial x,y
    400,400//initial width,height
    NULL,//handle to parent
    NULL,//handle to menu
    hinstance,//instance of this application
    NULL)))//extra creation parms
return (0);

//create the second window
if(!(hwnd = CreateWindowEx(NULL,//extended style
    WINDOW_CLASS_NAME,//class
    "Window 2 Also Based on WINCLASS1",//title
    WS_OVERLAPPEDWINDOW|WS_VISIBLE,
    100,100,//initial x,y
    400,400//initial width,height
    NULL,//handle to parent
    NULL,//handle to menu
    hinstance,//instance of this application
    NULL)))//extra creation parms
return (0);
```

当然，你可能希望用不同的变量而不是同一个变量来跟踪每一个窗口，就像 hwnd 中的例子那样，其实你已掌握了其要义，如一次打开两个窗口的例子。请看 DEMO2_5.CPP 以及 CD-ROM 上相关的可执行文件 DEMO2_5.EXE。当你运行.EXE 执行文件时，应当可以看到如图 2.13 所示的情况。注意当你关闭任何一个窗口时，两个窗口将同时关闭，并且应用程序也终止。看一看是否能够找到每次只关闭一个窗口的方法。(提示：创建两个 Windows 类，直到两个窗口都关闭之后再传递 WM_QUIT 消息。)

总结

尽管我并了解你，但是我还是很为你激动！到目前为止，你已具备了 Windows 编程的基本知识并需要开始掌握更复杂的 Windows 编程知识。你已了解了 Windows 和多任务的结构，并且也知道了如何创建一个 Windows 类、注册类、创建窗口、编写事件循环和句柄等等内容。因此你已经打下了坚实的 Windows 编程基础。你完成了一项最杰出的任务。

下一章中，我们将了解更多的和 Windows 相关的内容，如：使用资源、创建菜单、使用对话框以及获取信息等。

第三章 高级 Windows 编程

在火箭专家看来，Windows 编程当然算不上什么大工程。但 Windows 编程很绝的地方在于，你不用了解太多细节，就可以完成很多工作。因此，我们在本章中将主要讨论开发一个完整的 Windows 应用程序所需要的一些最重要的内容。本章主要内容有：

- 使用资源(如图标、光标和声音)
- 菜单
- 基本的图形设备接口和视频系统
- 输入设备
- 传递消息

使用资源

Windows 创建者提出一个主要设计目标就是，在一个 Windows 应用程序中除程序代码外还能储存更多的资源(甚至 Mac 程序也是如此)。他们认为一个程序的数据也能够驻留在该程序的.EXE 文件中。这是个不错的想法，因为：

- 同时含有代码和数据的.EXE 文件更容易分配。
- 如果没有外部数据文件的话，就会丢失这些数据。
- 外部强制转移不会很容易地访问、任意删改、添加、和分配你的数据文件（例如，.BMP 文件、.WAV 文件等等）。

要满足这种数据库技术，Windows 程序支持该种功能，称之为资源。这只是与你的程序代码结合在一起的数据的一小部分，这部分数据在以后的运行过程中可被程序本身加载。图 3.1 解释了这个概念。

图 3.1(略)

那我们讨论的是哪一种资源呢？实际上对于想编译进程序中的数据类型并没有什么限制，因为 Windows 程序支持用户定义的资源类型。但是应当注意几种预定义的类型：

- 图标——小位图图形，可以用于许多方面，例如单击该图形运行一个目录下的一个程序。图标使用.ICO 文件扩展名。
- 光标——一个表示鼠标指针的位图。Windows 允许以各种方式操作光标。例如，可以令光标在窗口之间移动时变换。光标使用.CUR 文件扩展名。
- 字符串——字符串资源对于作为一种资源来讲可能是最不明显的了。可以这样说：“我经常将字符串添加到我的程序中或者一个数据文件中。”我知道你的意思。然而，Windws 允许将一个字符串表作为一种资源放到你的程序中，并且通过标识符来访问它们。
- 声音——大部分 Windows 程序至少都可以通过.WAV 文件来使用声音。因此，.WAV 文件也是一种资源。
- 位图——这是标准的位图，可以是单色、4 位、8 位、16 位或 32 位格式的像素矩阵。在图形操作系统（如 Windows)中是非常常用的对象，因此也可以将位图作为一种资源。位图使用.BMP 文件扩展名。
- 对话框——对话框在 Windows 中也非常常用，设计者可以让对话框作为一种资源，而不是在外部装载的东西。好主意！因此，你可以在程序中创建对话框，

也可以将它们设计为一个编辑器，作为一种资源来存储。

• 图元文件——图元文件相对高级。它们允许将一个图像操作作为一个序列记录在一个文件中，然后再回放它。

现在你已经了解了资源的定义以及形式，下一步就是如何将它们一起使用。好！有一个资源编译器的程序，可以以一个扩展名为.RC 的 ASCI 文本资源文件输入。该文件是一个 C/English 文件——描述了编译到一个数据文件中的所有资源。该资源编译器装载所有的资源，以.RES 的扩展名形式将所有资源放置在一个大数据文件中。

这个.RES 文件包含了你在.RC 文件中定义的诸如图标、光标、位图、声音等所有资源的二进制数据。该.RES 文件和.CPP、.H、.LIB、.OBJ 等等文件一样都可以编译成一个.EXE 文件，这就已经足够了！图 3.2 显示了该过程的数据流程的可能性。

集合资源

以前可以使用一个外部资源编译器，如 RC.EXE 将所有的资源编译到一起。但是现在可以使用编译器 IDE 来做这些工作。因此，如果在程序中添加一种资源的话，可以简单地通过 IDE 中的“文件”菜单中选择“新建”按钮（大多数情况下），然后选择想要添加的资源类型（的面将详细讨论）来添加资源。

让我们回顾一下如何处理资源：可以向程序中添加许多数据类型和对象，然后它们以资源的形式和实际程序代码一起驻留在.EXE 文件中（一般在文件的末尾某处）. 在运行过程中，可以访问这个资源数据库，并且可以从程序本身（而不是作为一个单独的文件从磁盘中）装载资源数据. 要创建该资源文件，必须有一个以 ASCⅡ文本形式的资源描述文件，名称为.RC。然后将该文件传递到编译器中（一起访问该资源），并且产生一个.RES 文件。然后将该.RES 文件和所有的其他程序对象连接到一起，创建一个最终的.EXE 文件。就这么简单！好，这样我就变成了一个亿万富翁了。

记住上述所有内容，下面就让我们讨论一下众多的资源对象，学会如何创建并装载到程序中。我就不再重复上面提到过的所有的资源，但是你应当能够指出任何其他的信息对象。它们都以相同的方式运行，产生或采用一个数据类型、句柄或熬一个通宵不休息的精神插曲。

使用图标资源

使用资源只需要创建两个文件：一个是.RC 文件，如果想在.RC 文件中对符号标识符进行标注的话，可能还需要创建一个.H 文件。下面内容中将详细讨论。当然，最后还需要产生一个.RES 文件，但是我们让 IDE 编译器来做这个工作。

作为一个创建图标资源的例子，让我们看一下如何改变任务栏上的应用程序使用的图标以及窗口本身的系统菜单的图标。如果你还记得的话，我们在 Windows 类的创建过程中使用下面代码设置过这些图标：

```
winclass.hIcon = LoadIcon(NULL,IDI_APPLICATION);
winclass.hIconSm = LoadIcon(NULL,IDI_APPLICATION);
```

这两行代码为这些普通图标和小版本的图标加载默认的应用图标程序。实际上可以通过使用已经编译到一个资源文件中的图标，将所需要的图标装载到这些存取窗口中。

首先需要一个图标，我已经创建了一个很酷的图标用于本书中所有的应用程序。该图标名为 T3DX.ICO，如图 3.3 所示。我使用了 VC++5.0 的图形编辑器（如图 3.4 所示）创建了该图标。当然你可以使用任何你想使用的程序（只要该程序支持该种输出类型）来创建图标、光标和位图等等。

图 3.3 T3DX.ICO 图标位图(略)

T3DX.ICO 是一个 32 像素×32 像素，16 位色彩。图标可以安排在 16×16 到 64×64 的区域内，最高可以达到 256 色彩。但是大多数图标都是 32×32 以及 16 色彩的，我们也采用这种格式。

一旦你有了感兴趣的、需要放入一个资源文件中的图标，就需要创建一个资源文件来放置该图标。为了简单起见，需要人工编写。（请记住，IDE 编译器完全可以做这些工作，但是那样的话，你就什么也学不到了，不是吗？）

.RC 文件包含所有资源的定义，也就是说在程序中可以使用多种资源。

注意：在编写任何代码之前，我想指出关于资源的非常重要的一点，Windows 可以使用 ASCII 文本字符串或者是整数标识符来引用资源。在大多数情况下，你可以在.RC 文件中同时使用这两种方式，但是应当注意一些资源只允许使用其中的一种。无论是哪种情况，资源必须以稍微不同的方式来加载，并且如果到标识符的话，在你的工程中必须包含一个另外的符号前后对照的.H 文件。

下面是如何在.RC 脚本文件中定义一个 ICON 资源：

方法Ⅰ——使用字符串名

```
icon_name ICON FILENAME.ICO
```

例如：

```
windowicon ICON star.icon
MyCoolIcon ICON cool.ico
```

或者是

方法Ⅱ——通过整数标识符

```
icon_id ICON FILENAME.ICO
```

例如：

```
windowicon ICON star.ico
124 ICON ship.ico
```

这是令人混乱的地方：注意方法Ⅰ中根本没有任何注解。这就是个问题了，容易给你带来麻烦，因此要仔细听。应当能注意到：ICON 定义的每一种方法的第一个例子看上去完全一样。但是，一个理解为“windowicon”，而另一个符号 windowicon。这样就导致在.RC 文件中必须包含一个附加文件，来定义符号常量。当资源编译器解析下面代码时，

```
windowicon ICON star.ico
```

资源编译器首先看一下已经在 iclude 头文件定义的符号。如果该符号存在，资源编译器就通过整型标识符来引用该符号所指向的资源。否则，资源编译器就假定它是个字符串，通过字符器“windowicon”来引用 ICON。

因此，如果想在.RC 资源脚本中定义符号 ICON 的话，也需要一下.H 文件来解析该符号索引。要想在.RC 脚本中包含.H 文件，应当使用标准 C/C++#include 描述符。

例如，假定你想在.RC 文件 RESOURCES.RC 中定义三个符号图标，同是也需要一个.h 文件 RESOURCES.H。下面是每个文件的内容：

RESOURCES.H 的内容：

```
#define ID_ICON1 100 //these numbers are arbitrary
#define ID_ICON2 101
#define ID_ICON3 102
```

RESOURCES.RC 的内容：

```
#include "RESOURCES.H"
//here are the icon defines,note the use of C++ comments
ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

就是这样。然后可以将 RESOURCES.RC 添加到你的工程中，确认应用程序文件中有#include RESOURCES.H，然后你就大功告成了。当然，.ICO 文件必须放在工程的工作目录下，以便于资源编译器能够找到它们。

如果没有为图标定义（#define）符号，也没有包含一个.H 文件，资源编译器将只能假定符号 ID_ICON1、ID_ICON2 和 ID_ICON3 是字符串。然后就是如何在程序“ID_ICON1”、“ID_ICON2”、“ID_ICON3”中引用它们。

我已经完全打乱了所论及问题的时间/空间连续性，让我们回顾一下想做的工作——仅仅是加载一个简单的图标。

要想通过字符串名来装载一个图标，按下面步骤进行：

在.RC 文件中：

```
your_icon_name ICON filename.ico
```

在程序代码中：

```
//Notice the use of hinstance instead of NULL。
Winclass.hIcon = LoadIcon(hinstance,"your_icon_name)";(注：//原文如此)
Winclass.hIconSm = LoadIcon(hinstance,"your_icon_name)";
```

要通过符号参考来装载，可以如前面例子#include 那些包含符号参考的头文件。

在.H 文件中：

```
#define ID_ICON1 100 //these numbers are arbitrary
#define ID_ICON2 101
#define ID_ICON3 102
```

在.RC 文件中：

```
//here are the icon defines,note the use of C++ comments
ID_ICON1 ICON star.ico
ID_ICON2 ICON ball.ico
ID_ICON3 ICON cross.ico
```

程序代码可以像下面一样：

```
//Notice the use of hinstance instead of NULL。
//use the MAKEINTRESOURCE macro to reference
Winclass.hIcon = LoadIcon(hinstance,MAKEINTRESOURCE(ID_ICON1));
Winclass.hIconSm = LoadIcon(hinstance,MAKEINTRESOURCE(ID_ICON1));
```

注意宏 MAKEINTRESOURCE()的使用。该宏将整数转换为一个字符串指针，但是不必担心该操作——只在使用已经#define 的符号常数时应用它。

使用光标资源

光标资源几乎和图标资源相同。光标文件是一个小位图，扩展名为.CUR，可以在大多数 IDE 编译器中创建，或者使用单独的图像处理程序来创建。光标通常是 32×32 以及 16 位色彩的，最高可达 64×64 以及 256 色彩，甚至可以采用动画。

假定已经使用 IDE 或一个单独的绘图程序创建了一个光标文件，将它们添加到.RC 文件中以及通过程序来访问它们的步骤和图标的情况相似。要定义一个光标，使用.RC 文件中的光标关键字。

方法Ⅰ——通过字符串

```
cursor_name CURSOR FILENAME.CUR
```

例如：

```
windowcursor CURSOR crosshair.cur
MyCoolCursor CURSOR greenarrow.cur
```

或者是

方法Ⅱ——通过整型标识符

```
cursor_id CRUSOR FILENAME.CUR
```

例如：

```
windowcursor CURSOR bluearrow.cur
292 CURSOR redcross.cur
```

当然，如果想使用符号标识符，必须创建一个.H 文件以及符号的定义。

RESOURCES.H 的内容：

```
#define ID_CURSOR_CROSSHAIR 200//these mumbers are arbitrary
#define ID_CURSOR_GREENARROW 201
```

RESOURCES.RC 的内容：

```
#include "RESOURCES.H"
//here are the icon defines,note the use of C++ comments
ID_CURSO_CROSSHAIR CURSOR crosshair.cur
ID_CURSOR_GREENARROW CURSOR greenarrow.cur
```

没有任何原因说明一个资源数据文件为什么不能存在于其他目录中。例如 greenarrow.cur 可能存在于一个 CURSOR 目录的根目录下，像下面一样：

```
ID_CURSOR_GREENARROW CURSOR C:\CURSOR\greenarrow.cur
```

技巧：我已经为本章创建了一些光标.cur 文件。使用你的 IDE 来浏览一下这些光标文件，或者仅打开该目录，Windows 将通过其文件名显示每个光标文件的位图。

现在已经了解了如何向一个.RC 文件中添加一个光标资源，下面是按照字符串名从应用程序中加载该资源的程序代码。

在.RC 文件中：

```
CrossHair CURSOR crosshair.cur
```

在程序代码中：

```
//注意使用 hinstance 而不是 NULL
Winclass.hCursor=LoadCursor(hinstance,"crossHair");
```

要通过.H 文件中定义的符号标识来装载光标，具体步骤如下。

在.H 文件中：

```
#define ID_CROSSHAIR 200
```

在.RC 文件中：

```
ID_CROSSAIR CURSOR crosshair.cur
```

在程序代码中：

```
//注意使用 hinstance 而不是 NULL

Winclass.hCursor=LoadCursor(hinstance,MAKEINTRESSOURCE(ID_CROSSHAIR))
;
```

现在已经是第二次使用宏 MAKEINTRESOURCE()来将符号整型 ID 转换为 Windows 系统使用的格式。

好，有一个细节可能还没有引起你的注意。到现在为止只遇到了 Windows 类图标和光标。但是在窗口之间操作窗口图标和光标可能吗？例如，如果想创建二个窗口，并且使光标在两个窗口之间不同。要想做到这一点话，应当使用 SetCursor()函数：

```
HCURSOR SetCursor(HCURSOR hCursor);
```

其中，hCursor 是一个由 LoadCursor()检索光标句柄。使用该技术唯一的问题就是 SetCursor()函数不太灵便，因此应用程序中必须在鼠标从一个窗口移动到另一个窗口的同时跟踪和改变光标，下面是一个设置光标的例子：

```
//load the cursor somewhere maybe in the WM_CREATE
HCURSOR hcrosshair = LoadCursor(hinstance,"CrossHair");
//later in program code to change the cursor_
SetCursor(hcrosshair);
```

CD-ROM 上 DEMO3_1.CPP 给出了一个设置窗口图标和鼠标光标的例子。下面清单给出了加载新图标和光标的重要的代码部分的摘录。

```
//include resources
#include "DEMO3_1RES.H"
.
.
//changes to the window class definition
winclass.hIcon =
  LoadIcon(hinstance,MAKEINTRESOURCE(ICON_T3DX));
winclass.hCrusor=
  LoadCursor(hinstance,MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hIconSm = LoadIcon(hinstance,MAKEINTRESOURCE(ICON_T3DX));
```

并且，程序使用了资源脚本程序 DEMO3_1.RC 和资源头文件 DEMO3_1RES.H。

DEMO3_1RES.H 的内容：

```
#define ICON_T3DX 100
#define CURSOR_CROSSHAIR 200
```

DEMO3_1.RC 的内容：

```
#include "DEMO3_1RES.H"
//note that this file has defferent types of resources
ICON_T3DX ICON t3dx.ico
CURSOR_CURSSHAIR CURSOR crosshair.cur
```

要自己建立应用程序，你需要下面文件：

DEMO3_1.CPP——C/C++主文件
DEMO3_1RES.H——定义符号的头文件
DEMO3_1.RC——资源脚本程序
T3DX.ICO——图标的位图数据
CROSSHAIR.CUR——光标的位图数据

所有这些文件都应当放在同一个目录下作为一个工程。否则编译器和连接器很难找到它们。一旦创建并运行了该程序，或者使用预编译程序 DEMO3_1.EXE，就应当看到如图 3.5 所示的图像，非常酷，是吧！

----------图 3.5 关于常用图标和光标的 DEMO3_1.EXE 的输入结果----------

作为一个试验，尝试用 IDE 打开 DEMO3_1.RC 文件。图 3.6 表示了我使用 VC++5.0 所作的内容。但是，使用专用编译器应当得到不同的结果，因此如果结果不相同也不必大惊小怪。OK！在进入下面内容之前讨论一下 IDE。如前所述，可以使用 IDE 来创建.RC 和.H 文件，但是，应当阅读一下 IDE 的手册。

但是装载一个人工编写的.RC 文件还有一个问题，如果使用你的 IDE 保存该文件的话，毫无疑问 Windows 编译器认为在.RC 文件中造成了无数的注释、宏、定义(#define)以及其他垃圾数据。实际上该问题的本质是如果想编辑人工编写的.RC 文件的话，应当将.RC 文件作为文本为加载来进行编辑。这样编译器就不会将它作为一个.RC 文件来装载，但是应当注意只能是无格式的 ASCII 文本。

创建字符串表格资源

如引言中所提到的，Windows 支持字符串资源。和其他资源不同的是，只能使用一个字符串表单来容纳所有的字符串。

并且字符串资源不允许通过字符串来定义。因此，在.RC 文件中定义的所有字符串表单必须和符号引用常量以及相关的解释该引用的.H 头文件同时使用。

我依然不能确认我使用字符串资源的感受。使用字符串资源实际上和使用头文件相同，并且两种情况下——字符串或无格式头文件——都必须重新编译。因此，我认为没有使用它们的必要。但如果你确实不嫌麻烦，可以将字符串资源放入.DLL 文件中，主程序就不必要再重新编译它们。但是，我是一个技术人员，不是思想家，谁去关心这些问题？

要在.RC 文件中创建一个字符串表单，必须使用下面语法：

```
STRINGTABLE
```

```
{
ID_STRING1,"string 1"
ID_STRING2,"string 2"
.
.
}
```

当然，符号常量可以是任何东西，就像注释中的字符串一样。只有一条原则：每行字符串不能超过 255 个字符(包括常量本身在内)。

下面是一个包含可能在游戏菜单中使用的字符串表单的.H 和.RC 文件的例子。.H 文件包含有：

```
//常数值取决于你
#define ID_STRING_START_GAME 16
#define ID_STRING_LOAD_GAME 17
#define ID_STRING_SAVE_GAME 18
#define ID_STRING_OPTIONS 19
#define ID_STRING_SAVE_EXIT 20
```

.RC 文件中包含有：

```
//note the stringtable does not have a name since
//only one stringtable is allowed per .RC file
STRINGTABLE
{
ID_STRING_START_GAME "Kill Some Aliens"
ID_STRING_LOAD_GAME "Doneload logs"
ID_STRING_SAVE_GAME "Upload Data"
ID_STRING_OPTIONS "Tweak The Settings"
ID_STRING_SAVE_EXIT "Let's Bail!"
}
```

技巧：你几乎可以将所有的想添加的内容都放到字符串表单中，包括 printf()命令格式如%d、%s 等等。不能使用换行符"\n"，但是可以使用八进制换码顺序如\015 等等。

一旦创建了含有字符串资源的源文件，应当使用 LoadString()函数来装载某一个字符串，下面是 LoadString()函数的原型：

```
int LoadString(HINSTANCE hInstance,//handle of module withstring resource
    UNIT uID,//resource identifier
    LPTSTR lpBuffer,//address of buffer for resource
    int nBufferMax);//size of buffer
```

LoadString 返回一个所读字符的数量，或者在调用不成功时返回 0。下面是如何在游戏运行过程中调用和保存游戏字符串的函数：

```
//create some storage space
char load_string[80],//used to hold load game string
  save_string[80],//used to hold save game string
```

```
//load in the first string and check for error
if(!LoadString(hinstance,ID_STRING_LOAD_GAME,load_string,80)
{
  //there's an error
}//end if

//load in the second string and check for error
if(!LoadString(hinstance,ID_STRING_SAVE_GAME,save_string,80)
{
  //there's an error
}//end if

//use the string now
```

和前面一样，hinstance 是应用程序在 WinMain()中传递的实例。

上面已经包含了所有的字符串资源的用法。如果你能为它们找到更好的用处，请给我发一个 email:ceo@games3d.com。

使用声音.WAV 资源

到现在为止，或许你已经能够方便地使用资源脚本程序，又或许你已经非常厌烦以至于打算攻击我的 Web 页，来对我进行破坏了。请记住，那是 Microsoft 公司(http://www.microsoft.com)开发的内容。我仅仅是了解它而已。

好吧，我已经向你声明了不经常使用的否认声明，让我们继续下面的装载声音资源的内容！

大多数游戏都是用两种声音类型中的一种：

■数据化.WAV 文件

■MIDI.MID 文件

据我所知，Windows 的标准资源仅支持.WAV 文件，因此我就只分析如何创建.WAV 资源。当然如果不支持.MID 资源的话，你可能可能创建用户定义的资源类型。我们现在不讨论这个主题，但别个会加以讨论。

所需要的第一个事情就是一个.WAV 文件，它只是一下含有大量的 8 位或 16 位的一定频率脉冲的数字式数据波形。对于游戏声音效果来讲，典型的脉冲频率为 11MHz、22MHz 和 44MHz(CD 级品质)。该内容仍然和你无关，但我希望对该 内容概述一下，在我们学习 DirectSound 时，将会学习有数字采样理论和.WAV 文件的全部内容。现在，只讨论一下样本大小和频率。

我们假设你的磁盘上已经有了.WAV 文件，并且希望将该文件添加到一个源文件中，能够装载并且按照一定的程序播放。Let's go on！.WAV 文件的源类型就是 WAVE——这是令人惊奇的事。要将该文件添加到.RC 文件中，应当使用下面语法：

方法 1——通过字符串名

```
wave_name WAVE FILENAME.WAV
```

例如：

```
BigExplosion WAVE exp11.wavr
FireWeapons WAVE fire.wav
```

方法 2——通过整型标识符

```
ID_WAVE CURSOR FILENAME.CUR
```

例如：

```
DEATH_SOUND_ID WAVE die.wav
20 WAVE intro.wav
```

当然，该符号常量应当在一个.H 文件中的某处进行定义，这部分内容我们已经有所了解。

对于这一点，我们可能碰上了一个小障碍：WAVE 资源要比光标、图表和字符串表单要复杂一点。所以问题是装载声音资源的程序要比装载其他资源要复杂一些，因此我们现在就不介绍在一个实际游戏中装载.WAV 资源的方式，在后面再详细介绍。现在介绍一下使用 PlaySound()函数装载和播放.WAV 文件的技巧。下面是 PlaySound()函数的原型：

```
BOOL PlaySound(LPCSTR pszSound,//string of sound to play
    HMODULE hmod,//instance of application
    DWORD fdwSound);//flags parameter
```

和 LoadString()不同的是，PlaySound()稍微有点复杂，因此我们要深入了解一下每一个参数：

■PszSound——该参数或者是资源文件中声音资源的字符串名，或者是磁盘上的文件名。并且可以使用 MAKEINTRESOURCE()并且应用使用符号常量定义的一个 WAVE。

■Hmod——装载该资源的应用程序实例。它只是一个应用程序实例。

■FdwSound——这是个关键参数。该参数控制声音如何装载和播放。表 3.1 列出了 FdwSound 的一些最有用的值。

表 3.1 PlaySound()函数的 FdwSound 参数的值

值	描述
SND_FILENAME	该 PszSound 参数是文件名
SND_RESOURCE	该 PszSound 参数是一个资源标识符：hmod 必须辨别包含该资源的实例
SND_MEMMORY	装载到 RAM 中的声音事件文件。FdwSound 参数指定的该参数必须指向内存中声音文件的图像
SND_SYNC	声音事件的同步反馈。声音事件播放完毕后，PlaySound()返回。
SND_ASYNC	声音事件的异步播放，开始播放声音后，PlaySound()立即返回。要终止异步播放的波形声音，调用 PlaySound()，并且 PszSound 设为.NULL

SND_LOOP	声音重复播放直到再次调用 PlaySound()，并且 PszSound 设为 NULL，并且需要指定 SND_ASYNC 标识符来制定一个异步声音事件
SND_NODEFAULT	不使用默认声音事件。如果没有发现声音文件，PlaySound()返回静音而不播放默认声音
SND_PURGE	因为调用任务而终止声音。如果 PszSound 不为 NULL 的话，将停止所有指定声音的事件。如果 PszSound 为 NULL 的话，将停止所有代表调用任务的声音
SND_NOSTOP	指定声音事件将让位于另外一个已经播放的声音事件。如果一个声音由于产生该声音所需要的资源正在播放其他声音文件而不能播放当前声音，该函数不播放所要求的声音，而是立即返回错误
SND_NOWAIT	如果驱动器正忙，不播放声音，该函数就立即返回

要使用 PlaySound()播放一个 WAVE 声音资源，一般需要下面四个步骤：

1.创建.WAV 文件并存储在磁盘上。

2.创建.RC 资源脚本程序以及相关的头文件。

3.编译该资源和程序代码。

4.使用 MAKEINTRESOURCE()宏，通过 WAVE 资源名或者是通过 WAVE 资源标识符在程序中设定一个 PlaySound()的调用。

让我们看几个例子。首先是有两种声音的常规 RC 文件：一个是字符串名的声音文件,另一个符号常量的声音文件,分别命名为 RESOURCE.RC 和 RESOURCE.H。该文件如下：

RESOURCE.H 文件包含：

```
#define SOUND_ID_ENERGIZE 1
```

RESOURCE.RC 文件包含：

```
#include "RESOURCE.H"
//first the string name defind sound resource
Telporter WAVE teleport.wav
//and now the symbolically defined sound
SOUND_ID_ENERGIZE WAVE energize.wav
```

在程序中，下面显示了如何以不同方式播放声音：

```
//to play the telport sound asynchronously
PlaySound("Teleporter",hinstance,
    SND_ASYNC|SND_RESOURCE);
//to play the telport sound asynchronously with looping
PLaySound("Teleporter",hinstance,
    SND_ASYNC|SND_LOOP|SND_RESOURC);
//to play the energize sound asynchronously
PlaySound(MAKEINTRESOURCE(SOUND_ID_ENERGIZE),hinstance
    SND_ASYNC|SND_RESOURCE);
//and if you simply wanted to play a sound off disk
```

```
//directly then you could do this
PlaySound("C:\path\filename.wav",hinstance,
    SND_ASYNC|SND_FILENAME);
```

要停止所有的声音，使用 SND_PURGE 标识符并将声音名设为 NULL，如下所示：

```
//stop all sounds
PlaySound(NULL,hinstance,SND_PURGE);
```

很明显，有许多标识符选项可以自由选择匹配。但是仍然没有任何控制或菜单，所有很难对演示应用程序产和影响。作为一个简单的使用声音资源的演示程序来讲，我已经创建了 DEMO3_2.CPP，可以从磁盘上找到。下面就将该程序列出清单，但是 99%的程序都是曾经使用过的标准的模板，而声音代码也和前面例子中的代码行基本相同。该演示程序是预编译的程序，可以运行 DEMO3_2.EXE 来浏览。

但是我还是列出所使用的.RC 文件和.H 文件，分别是 DEMO3_2.RC 和 DEMO3_2RES.H 文件：

DEMO3_2RES.H 文件内容：

```
#defines for sound ids
#define SOUND_ID_CREATE 1
#define SOUND_ID_MUSIC 2

//defines for icons
#define ICON_T3DX 500

//defines for icons
#define CURSOR_CROSSHAIR 600
```

DEMO3_2.RC 文件内容：

```
#include "DEMO3_2RES.H"
//the sound resources
SOUND_ID_CREATE WAVE create.wav
SOUND_ID_MUSIC WAVE techno.awv
//icon resources
ICON_T3DX ICON T3DX.ICO
//cursor resources
CURSOR_CROSSHAIR CURSOR CROSSHAIR.CUR
```

可以看到我们在代码中也包含了图标和光标资源，从而使程序更具趣味性。

要编制 DEMO3_2.CPP，我采用了标准 Windows 演示程序，并且在两部分内容中添加了声音代码的调用：WM_CREATE 消息和 WM_DESTROY 消息。在 WM_CREATE 消息处，设置了两个声音效果，一个声音在说“创建窗口(Creating window)”，然后停止：另一个是以循环模式播放的一小段音乐，能够一直播放下去。在 WM_DESTROY 消息部分停止所有的声音。

注意：第一段声音，我使用 SND_SYNC 标志。使用该标志是因为 PlaySound()一次只允许播放一个声音，而且在播放过程中，我不希望第二段声音终止第一段声音。

下面是从 DEMO3_2.CPP 文件中添加到 WM_CREATE 消息和 WM_DESTROY 消息中的代码：

```
case WM_CREATE:
{
  // do initialization stuff here

  // play the create sound once
  PlaySound(MAKEINTRESOURCE(SOUND_ID_CREATE),
            hinstance_app, SND_RESOURCE | SND_SYNC);

  // play the music in loop mode
  PlaySound(MAKEINTRESOURCE(SOUND_ID_MUSIC),
            hinstance_app, SND_RESOURCE | SND_ASYNC | SND_LOOP);

  // return success
  return(0);
} break;

case WM_DESTROY:
{
  // stop the sounds first
  PlaySound(NULL, hinstance_app, SND_PURGE);

  // kill the application, this sends a WM_QUIT message
  PostQuitMessage(0);

  // return success
  return(0);
} break;
```

从上面代码中，可以发现有一个变量 hinstance_app，用来作为 PlaySound()调用的应用程序实例句柄。这只是一个全局变量，用来保存 WinMain()中传递的实例。它的代码紧跟在 WinMain()中类定义的后面，如下所示：

```
.
.
//在全局变量中保存实例
hinstance_app = hinstance;
//注册该窗口类
if(!RegisterClassEx(&winclass))
  return(0);
.
.
```

要建立该应用程序，在工程中需要包含下列文件：

DEMO3_2.CPP——源文件主程序。

DEMO3_2.RES.H——包含所有符号的头文件。

DEMO3_2.RC——源文件脚本程序

TECHNO.WAV——音乐片断，需要放在工作目录下。

CREATE.WAV——创建窗口声音，需要放在工作目录下。

WINMM.LIB——Windows 多媒体库扩展。该文件位于编译器的 LIB\目录下。应当将该文件添加到从该目录到输出的所有工程中。

MMSYSTEM.H——WINMM.LIB 的头文件，该文件已被包含在 DEMO3_2.CPP 和我的所有演示程序中。你所应了解的就是该文件应当位于你的编译器搜索路径中。它也是标准 Win32 头文件集中的一部分。

使用编辑器创建.RC 文件

创建 Windows 应用程序的大多数编译器都需要一个相当大的开发环境。如 Windows 的 Visual Development Studio 等。每一个 IDE 都含有一个或多个使用即插即用技术的自动创建不同资源、资源脚本程序和相关头文件的工具。

使用这些工具的唯一问题是要学习这些工具！并且，使用 IDE 创建的.RC 文件都是人工可读写的 ASCII 码文本，但是又有大量的添加的定义(#define)和宏，编译器能够添加它们，以便于自动完成并且简化常量的选择和 MFC 界面工作。

因为现在我是 Microsoft VC++5.0 用户，我简单说明一下关于使用 VC++5.0 资源处理支持的关键要素。首先，向工程中添加资源可以有两种方式：

方法 1——从主菜单中使用 File、New 选项，可以向工程中添加大量的资源。图 3.7 就是进行该操作后产生的对话框屏图。当添加图标、光标、位图等资源时，IDE 编译器自运行生成 Image Editor(如前面图 3.4 所示)。这是一个原始的图像编译单元，可以用来绘制光标和图标。如果要添加一个菜单资源（该部分内容将在下面部分讨论），就会出现菜单编辑器。

图 3.7 使用 VC++5.0 中的文件、新建来添加资源

方法 2——该方法更灵活一点，含有所有可能的资源类型，而方法 1 只支持其中的一部分。要向工程中添加任何一种类型的资源，可以使用主菜单中的 Insert、Resource 选项。3.8 表示了显示出的对话框。在这种情况下，你还需要进行一些维护（手工进行编辑）。无论何时想添加一个资源，都必须将它添加到资源脚本程序中——对吗？因此，如果你的工程中还没有一个资源脚本程序的话，编译器 IDE 将为你产生一个脚本程序，你之为 SCRIPT*.RC。另外，这两种方法最终都将生成一个名为 RESOURCE.H 的文件。该文件含有使用编辑器定义的和资源有关的资源符号、标识符值等。

我当然希望深入探讨有关使用 IDE 进行资源编辑的内容，但由于它只是整章内容的一部分，而不是一整本书，所以你自己阅读一下关于使用编译器进行文档编制的内容。本书中我们将不会使用更多的资源，因此上述讨论的信息已经足够了。下面让我们进行更复杂的资源类型——菜单的学习吧。

使用菜单编程

菜单是 Windows 程序中最酷的一个内容，可以说是人机交互式界面的关键所在（例如制造一个文字处理器）。了解如何创建和使用菜单是非常重要的，这是因为你可能想制作简单的工具来帮助创建游戏，或者是想有一个视窗基础的前端来作为游戏的开始。而这些工具毋庸置疑地要有大量的菜单（如果要制作一个

3D 工具的话，可能需要上百万个菜单）。请相信我！无论是哪种情况，都必须掌握如何创建、装载和响应菜单。

创建一个菜单

使用编译器的菜单编辑器可以创建一个完整的菜单以及相关的文件，但是现在我们人工编写它，因为我不知道你在使用哪种编译器，这样你也可以学会菜单说明书中的内容。但是当你正在编写一个实际的应用程序创建一个菜单时，大多数是应用 IDE 编辑器来创建菜单，因为人工输入编写菜单实在是太复杂了。它就像 HTML 代码——当启动 Web 后，使用一个文本编辑器来制作一个主页不是一件很复杂的事。但是，不使用任何工具来创建 Web 站点几乎是不可能的。

现在就让我们开始制作菜单！实际上菜单和已经讨论过的其他资源完全一样。它们驻留在一个.RC 资源脚本程序中，并且必须拥有一个.H 文件来解决符号引用问题，符号引用是所有菜单的标识符（一个例外：菜单名必须是符号的，无名称字符串）。下面是在.RC 文件中常见的一个菜单说明的基本语法：

```
MENU_NUME MENU DISCARDABLE
{//you can use BEGIN instead of { if you wish
//menu definitions
}//you can use END instead of } if you wish
```

MENU_NAME 可以是一个名称字符串或这是一个符号，关键字 DISCARDABLE 是不完整的但又是必须的。看上去非常简单，当然，中间内容省略了，我会在的后讨论！

在编写定义菜单项和子菜单的代码之前，我们首先了解一些直观的相关术语。上述讨论可能参见图 3.9 中菜单，它有两个一级菜单：File 和 Help。File 菜单中包含四个菜单项：Open、Close、Save、和 Exit。帮助菜单中只有一个菜单项：About。因此说本菜单中含有一级菜单和菜单项。但是，这容易令人造成误解，因为菜单中还可能有菜单或者是层叠式菜单。我不准备创建层叠式菜单，但是层叠式菜单的原理很简单：使用一个菜单定义来定义一个菜单项本身，并且可以循环无休止的这样做下去。

图 3.9 有两个子菜单的菜单栏

现在我们了解相关术语，下面是执行图 3.9 所示的菜单：

```
MainMenu MENU DISCARDABLE
{
POPUP "File"
  {
    MENUITEM "Open", MENU_FILE_ID_OPEN
    MENUITEM "Close", MENU_FILE_ID_CLOSE
    MENUITEM "Save", MENU_FILE_ID_SAVE
    MENUITEM "Exit", MENU_FILE_ID_EXIT
  }// end popup
POPUP "Help"
  {
```

```
    MENUITEM "About", MENU_HELP_ABOUT
    }// end popup
}//end top level menu
```

我们来逐段分析该菜单定义。首先该菜单名为 MainMenu。对于这一点，我们并不知道它是一个名称字符串，还是一个标识符，但是因为我一般都大写所有常量的第一个字母，很清楚它是一个无格式字符串。这也正是我要做的内容。向下看，有两个一级菜单定义，都以关键字 POPUP 开关——这就是关键所在。POPUP 指出了一个菜单可以使用下面 ASCII 名称和菜单项来定义。

ASCII 名必须跟在关键字 POPUP 后面，并且用括号来包起来。弹出式菜单定义必须放在{}或 BEGIN END 块中——喜欢使用哪种都可以。（使用 Pascal 语言的人应当高兴了。）

在该定义块中，后面是所有的菜单项。一定义一个菜单项，应当以下面语法使用关键字 MENUITEM:

```
MENUITEM "name", MENU_ID
```

就是这样。当然在该例子中，没有发现任何符号，但是可以在.H 文件中看到如下所示的格式：

```
//defines for the top level menu FILE
#define MENU_FILE_ID_OPEN 1000
#define MENU_FILE_ID_CLOSE 1001
#define MENU_FILE_ID_SAVE 1002
#define MENU_FILE_ID_EXIT 1003
//defines for the top level menu HELP
#define MENU_HELP_ABOUT 2000
```

提示：注意标识符的值，我选择 1000 为第一个一级菜单的开始，然后每个菜单项加 1。下一个一级菜单在 1000 基础上再加 1000.由此每一个最高级别的菜单间相差 1000，菜单中的每一个菜单项相差 1。这是一种良好的工作习惯，并且很少填充。

我没有定义“MainMenu"，因为我希望通过字符串而不是标识符来访问该菜单。这并不是唯一的方式。例如，假如我在.H 文件中将一行代码和符号放在一起，

```
#define MainMenu 100
```

资源编译器将自动假定我希望通过标识符来访问该菜单。我就必须使用 MAKEINTRESOURCE(MainMenu)或 MAKEINTRESOURCE(100)来访问该菜单资源。知道了吗？好，继续吧!

技巧：注意许多菜单项才有热键或快捷键，可以不必使用鼠标手动选择一级菜单或菜单项。可以使用&符号表达到目的。所要做的工作就是在 POPUP 菜单或 MENUITEM 字符串中将该符号&放在想标识热键或快捷键的符号前面。例如：

```
MENUITEM "E&xit",MENU_FILE_ID_EXIT
```

这样 X 就成为热键，

```
POPUP " &File"
```

通过 AST+F 使得 F 成为一个快捷键。

现在已经了解了如何创建和定义一个菜单，让我们看一看如何将它装载到应用程序中以及如何连接到一个窗口。

装载一个菜单

有许多方法可以将一个菜单连接到一个窗口上。可以将一个菜单和 Windows 类中的所有窗口建立联系，或者将不同的菜单连接到创建的每一个窗口上。首先，我们讨论一下如何将一下菜单和 Windows 类本身建立联系。

在 Windows 类的定义中，有一行代码可以定义菜单：

```
winclass.lpszMenuName = NULL;
```

你的要做的工作就是将它赋值为该菜单资源的名称。好，下面就是如何赋值：

```
Winclass.lpszMenuName = "MainMenu";
```

如果“MainMenu”是一个常量的话，可以按下面方式做：

```
Winclass.lpszMenuName = MAKEINTRESOURCE(MainMenu);
```

这样做唯一的一个问题就是创建的每个窗口都会有这样一个相同的菜单。要解决该问题，可以在创建菜单过程中通过传递菜单的句柄来将一个菜单指定给一个窗口。但是，要使用菜单句柄，必须使用 LoadMenu()装载菜单资源。下面是 LoadMenu()的原型：

```
HMENU LoadMenu(HINSTANCE hInstance//handle of application instance
                LPCTSTR lpMenuName);//menu name string or
menu-resource identifier
```

如果函数调用成功的话，LoadMenu()向菜单资源返回一个 HMENU 句柄，这样就可以使用了。

下面是标准的 CreateWindow()函数调用，将菜单“MainMenu”装载到该菜单句柄参数中：

```
//create the window
if(!(hwnd = createWindowEx(NULL,//extended style
                           WINDOW_CLASS_NAME,//class
            "Sound Resource Demo ",//title
            WS_OVERLAPPEDWINDOW|WS_VISIBLE,
            0,0//initial x,y
            400,400//initial width,height
            LoadMenu(hinstance,"MainMenu"),//handle to menu
            hinstance,//instance of this application
            NULL)))//extra creation parms
  return(0);
```

如果 MainMenu 是一个常数，调用就和下面相似：

```
LoadMenu(hinstance,MAKEINTRESOURCE(MainMenu)),//handle to menu
```

提示：你可能会认为我太关注于资源是通过字符串还是通过符号常量来定义的区别了。但考虑到该问题是导致 Window 程序员自我毁坏的最多的原因，我认为它值得做更多的工作，不是吗？

当然，在.RC 文件中可以有许多不同的菜单，因此可以将一个不同的菜单连接到每一个窗口上。

将一个菜单连接到每一个窗口的最后一个方法是调用函数 SetMenu()：

```
BOOL SetMenu(HAND hWnd,//handle of widow to attach to
              HMENU hMenu);//hanele of menu
```

SetMenu()采用窗口句柄和菜单句柄（从 LoadMenu()中获取），直接将该菜

单连接到一个窗口上。这个菜单将侍于任何以前连接的菜单。下面是一个例子的清单，假设 Windows 类定义该菜单为 NULL，如同调用 CreateWindow()的菜单句柄一样：

```
//first fill in the window class structure
winclass.cbSize=Sizeof(WNDCLASSEX);
winclass.style = CS_DBLCLKS|CS_OWNDC|
CS_HREDRAW|CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance = hinstance;
winclass.hIcon = LoadIcon(hinstance,
                          MAKEINTRESOURCE(INCON_T3DX));
winclass.hCursor = LoadCursor(hinstance,
                          MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hbrBackground = GetStockObject(BLACK_BRUSH);
winclass.IpszMenuName = NULL;//note this is null
winclass.lpszClassName = WINDOW_CLASS_NAME;
winclass.hIconSm = LoadIcon(hinstance,MAKEINTRESOURCE(ICON_T3DX));
// register the window class
if (!RegisterClassEx(&winclass))
  return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL, // extended style
  WINDOW_CLASS_NAME, // class
  "Menu Resource Demo", // title
  WS_OVERLAPPEDWINDOW | WS_VISIBLE,
  0,0, // initial x,y
  400,400, // initial width, height
  NULL, // handle to parent
  NULL, // handle to menu, note it's null
  hinstance,// instance of this application
  NULL))) // extra creation parms
return(0);
//Since the window has been created you can
//attach a new at any time
//load the menu reource
HMENU hmenuhandle = loadMenu(hinstance,"MainMenu");
//attach the menu to the window
SetMenu(hwnd,hmenuhandle);
```

使用第二种方法创建菜单和将该菜单连接到一个窗口上的例子（也就是说在该窗口的创建调用过程中），可以参见 CD-ROM 上的 DEMO3_3.CPP 和相关的可执

行文件 DEMO3_3.EXE，图 3.10 表示了 DEMO3_3.EXE 正在运行的情况。

图 3.10 正在运行 DEM3_3.EXE

这里唯一需要注意的两个文件是资源文件和头文件：DEMO3_3RES.H 和 DEMO3_3.RC。

DEMO3_3RES.H 的内容：

```
// defines for the top level menu FILE
#define MENU_FILE_ID_OPEN 1000
#define MENU_FILE_ID_CLOSE 1001
#define MENU_FILE_ID_SAVE 1002
#define MENU_FILE_ID_EXIT 1003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT 2000
```

DEMO3_3.RC 的内容：

```
#include "DEMO3_3RES.H"

MainMenu MENU DISCARDABLE
{
  POPUP "File"
  {
    MENUITEM "Open", MENU_FILE_ID_OPEN
    MENUITEM "Close", MENU_FILE_ID_CLOSE
    MENUITEM "Save", MENU_FILE_ID_SAVE
    MENUITEM "Exit", MENU_FILE_ID_EXIT
  } // end popup

  POPUP "Help"
  {
    MENUITEM "About", MENU_HELP_ABOUT
  } // end popup

} // end top level menu
```

要编译自己的 DEMO3_3.CPP 可执行文件，一定要确认包含下面文件：

DEMO3_3.CPP——主要源程序。

DEMO3_3RES.H——符号头文件源程序。

DEMO3_3.RC——资源脚本程序。

尝试运行 DEMO3_3.EXE 和相关源程序。改变菜单项，通过向.RC 文件中添加更多 POPUP 块来添加更多的菜单。并且尝试创建一个层叠式的菜单树。（提示：对于创建菜单的 MENUITEM 来说，只是用一个 POPUP 块来替换 MENUITEM）。

对菜单事件消息的响应

DEMO3_3.EXE 的唯一问题是它不能做任何事情。的确，其中主要问题就是不知道如何侦测菜单项选择和操作产生的消息。本节将主要讨论这个主题。

当滑过一级菜单项时，Windows 菜单系统产生大量的消息（如图 3.11 所示）。

图 3.11 Windows 菜单选择消息流

我们感兴趣的是在选中一个菜单项后放开鼠标时的消息传递。这指的是一个选择过程。选择过程将一个 WM_COMMAND 消息传递到该菜单连接的窗口 WinProc()函数中，指定的菜单项标识符和其他各种数据存储在该消息的 wparam 和 lparam 中，如下所示：

msg——WM_COMMAND。

lparam——传递消息的窗口句柄。

wparam——选中的菜单项标识符。

提示：从技术角度上讲，应当从 wparam 中提取低位的 WORD，以保证 LOWORD()宏安全。该宏是标准包括中的一部分。因此必须要阅读该内容。

因此你只需要调用函数 switch()来处理 wparam 的参数值，菜单中定义的另一种 MENUITEM 标识符，就是你的工作了。例如，使用 DEMO3_3.RC 文件中定义的菜单，还应当添加 WM_COMMAND 消息句柄，WinProc()以下面方式结束：

```
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam)
{
  // this is the main message handler of the system
  PAINTSTRUCT ps; // used in WM_PAINT
  HDC hdc; // handle to a device context

  // what is the message
  switch(msg)
  {
    case WM_CREATE:
    {
      // do initialization stuff here

      // return success
      return(0);
    } break;
    case WM_COMMAND:
    {
      switch(LOWORD(wparam))
      {
        //handle the FILE menu
        case MENU_FILE_ID_OPEN:
        {
          //do work here
        }//break;
```

```
      case MENU_FILE_ID_CLOSE:
      {
        //do work here
      }//break;
      case MENU_FILE_ID_SAVE:
      {
        //do work here
      }//break;
      case MENU_FILE_ID_EXIT:
      {
        //do work here
      }//break;
      //handle the HELP menu
      case MENU_HELP_ABOUT:
      {
        //do work here
      }//break;
      default:break;
    }//end switch wparam
  }break;//end WM_COMMAND

  case WM_PAINT:
  {
    // simply validate the window
    hdc = BeginPaint(hwnd,&ps);
    // you would do all your painting here
    EndPaint(hwnd,&ps);

    // return success
    return(0);
  } break;

  case WM_DESTROY:
  {
    // kill the application, this sends a WM_QUIT message
    PostQuitMessage(0);

    // return success
    return(0);
  } break;

  default:break;

} // end switch
```

```
    // process any messages that we didn't take care of
    return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

就是如此简单，但这其实是非法的！当然还有其他的操作一级菜单和菜单项本身的消息，可以阅读 Win32 SDK 帮助来获得更多的信息。（我几乎无需了解除一个菜单项是否被单击以外的信息。）

作为菜单工作内容的现实例子，我已经创建了一个很不错的声音演示程序，可以允许通过主菜单来结束一个程序，演示四种不同端声音效果中的一个，最后通过“Help”菜单弹出一个 Help 对话框。并且.RC 文件中包含声音、图表和光标资源。该程序就是 DEMO3_4.CPP，让我们首先看一下其资源脚本程序和头文件。

DEMO3_4RES.H 的内容：

图形设备接口 GDI 介绍
处理重要事件
将消息传递给自己
总结

第四章 WindowsGDI、控件和突发奇想

本章是纯粹的 Windows 编程内容的最后一章。我们将详细讨论使用图形设备接口界面的内容。内容包括绘制像素、线和简单的几何开关。然后我们简略讨论一下定时方法，以 Windows 的子控件来结束 Windows 编程内容。最后简要介绍一下 T3D 游戏控制台模板程序，在本书后面所有内容中将该模板程序作为所有演示的开始。下面列出了本章涉及的主要内容：

- 高级 GDI 编程、笔、画刷和渲染
- 子控件
- 系统定时函数
- 传递消息
- 获取消息
- T3D 游戏控制台

高级 GDI 图形

我曾经提到过，GDI 和 DirectX 相比太慢了。但是 GDI 在各方面都很优秀，并且是 Windows 系统内置的渲染引擎，也就是说，如果想创建任何工具或标准的 GDI 应用程序了解关于 GDI 的工作方式是很有益处的，而了解如何将 GDI 和 DirectX 混全合一起使用则是使用 GDI 功能来模拟 DirectX 程序中未能实现的函数一种好方法。因此 GDI 可以在编写游戏程序演示功能时作为一种速度较慢的软件仿真，至少你还可以了解 GDI 的内容。

下面我们来讨论一下一些基本的 GDI 操作，还可以通过浏览 Win32 SDK 来了解更多的 GDI 内容，在本书中可以学到一些基本的技巧，而不是 GDI 的每一个功能。这就像看一个计算机分销商促展览——看到了一个，也就看到全部。

图形设备描述表

在第三章“高级 Windows 编程中”，可以经常看到设备描述表的类型句柄，或者是 HDC 即表示设备描述表的句柄的数据类型。这里设备描述表是一个图形设备描述表类型，当然还有其他的设备描述表，如打印机设备描述表。无论怎样讲，可能都对什么是图形设备描述表，以及图形设备描述表到底代表什么意思而感到疑惑不解。这都是很好的问题。

一个图形设备描述表实际上就是对一种安装在系统中的视频图像卡的描述。因此，当访问一个图形设备描述表或句柄时，实际上就表示安装在计算机系统上的视频卡具体描述及其分辨率和色彩容量。对于使用 GDI 的任何图形调用，该信息都是必须的。从本质上说，你所提供的指向任何 GDI 函数的 HDC 句柄，都用来访问一个函数需要操作的视频系统的重要信息。这就是需要一个图形设备描述表的原因所在。

图形设备描述表跟踪编程过程中可能改变的软件设置。例如，GDI 使用大量的图形对象，如笔、画刷、线和类型等等。GDI 使用上述基本数据描述来绘制任何一种图元。因此尽管当前画笔颜色是你设置的某种颜色，并且也不是视频卡的默认颜色，但是图形设备描述表仍然跟踪它。因此，图形设备描述表不仅是视频系统的硬件描述，而且还是记录和保存设置的信息库，由此你的 GDI 函数调用能

够使用这些设置，而不是和这些调用一起发送。这样可以保存 GDI 调用的大量参数。下面让我们看一下如何使用 GDI 对图形渲染。

色彩、画笔和画刷

如果认真考虑的话，能够在计算机屏幕上绘制的对象的类型并没有多少。当然绘制图形的形状和颜色有无穷多种，但对象的类型是很有限的。这些对象就是点、线和矩形。其他的任何东西都是这些基本图元对象类型的组合。

GDI 所采用的这种方法有点像一个画家。一个画家使用颜色、画笔和画刷来绘画——我们也是这样。GDI 以相同的方式工作，并且还有下面的定义：

- **画笔**——用于画线条和轮廓，具有颜色、粗细和线型。
- **画刷**——用于填充任何闭合的对象，具有颜色、样式、甚至可以是位图、

图 4.1 绘出了一个详细的标签。

在我们具体接触画笔和画刷以及实际使用它们之前，必须先了解 GDI 的情况。GDI 一般一次只使用一个画笔和一个画刷。在你的系统配置中可以有许多画笔和画刷，但是在当前图形设备描述表中每次只有一个画笔或画刷被激活。这样就必须为图形设备描述表选择对象，以便于使用。

请记住，图形设备描述表不仅是一个视频卡及其工作的描述，而且还是当前绘制工具的描述。画笔和画刷是该设备描述表跟踪的工具中的主要的样本，并且必须从该图形设备描述表中或之外选择这些工具。该过程称之为选定。当程序运行时，将选定一个新的画笔，然后选择画笔输出，也可能选定不同的画刷并且选择画刷输出等等。应当记住的是一旦该内容中选定了一个绘图对象，就要一直使用该对象，直到改变对象为止。

最后，无论何时创建了一个新的画笔和画刷，在完成绘制图形之后必须删除该画笔或画刷。这是非常重要的，因为 Windows GDI 关于画笔和画刷句柄就有如此之多的存取窗口，通常学会 GDI 几乎要耗尽精力，但我们一会儿就可以学会，好，现在我们首先讨论画笔，然后是画刷。

使用画笔

画笔句柄称这为 **HPEN**。下面是如何创建一个 NULL 画笔：

```
HPEN pen_1 = NULL;
```

pen_1 指示一个画笔句柄，但是，pen_1 仍然不能使用所希望的信息来替代或删除。该操作可以通过下面两种方法中的一种来完成。

- 使用存储对象
- 创建一个用户定义的画笔

请记住，存储对象或者是存储任何东西仅仅是 Windows 所拥有的一些默认样式的对象。对于画笔，已经有许多已经定义的画笔样式，但是这还是有限的样式。可以使用下面所示的 GetStockObject()函数来检索大量的不同的对象句柄，包括画笔句柄、画刷等等。

```
HGDIOBJ GetStockObject(int fnObject);//type of stock object
```

该函数只采用希望的存储对象的样式，返回一个该对象的句柄，画笔的样式是预选定义的存储对象，如表 4.1 所示。

表 4.1 存储对象样式

值	描述
BLACK_PEN	黑色画笔
NULL_PEN	中空的画笔
WHITE_PEN	白色画笔
BLACK_BRUSH	黑色画刷
DKGRAY_BRUSH	深灰色画刷
GRAY_BRUSH	灰色画刷
HOLLOW_BRUSH	中空的画刷(相当于 NULL_BRUSH)
LIGRAY_BRUSH	淡灰色画刷
NULL_BRUSH	中空的画刷（相当于 HOLLOW_BRUSH）
WHITE_BRUSH	白色画刷
ANSI_FIXED_FONT	标准的 Windows 固定间距（等宽）的系统字体
ANSI_VAR_FONT	标准的 Windows 可变间距（成比例间距）的系统字体
DEFAULT_GUIFONT	只用于 Windows95：用户界面对象如菜单和对话框的默认字体
OEM_PIXED_FONT	由生产商（OEM）确定的固定间距（等宽）字体
SYSTEM_FONT	系统字体，默认情况下，Windows 使用系统字体来绘制菜单，对话框控制功能和文本。在 Windows3.0 版本之后的系统中，系统字体为成比例间距字体，3.0 以前的 Windows 版本使用等宽系统字体
SYSTEM_FIXED_FONT	Windows3.0 之前的版本使用的固定间距（等宽）系统字体。该存储对象用来和 Windows 早期的版本兼容

由表 4.1 可以看到，并不能从表 4.1 中选择全部的画笔样式(这只是 GDI 的一点幽默)。下面是如何创建一个白色画笔的例子：

```
HPEN white_pen = NULL;
White_pen = GetStockObject(WHITE_PEN);
```

当然 GDI 并不知道 white_pen，因为并不能将 white_pen 选定到图形设备描述表中，但我们将它选定了。

创建画笔的更有趣的方法是通过定义画笔颜色、线条样式和像素宽度来自己创建画笔。用来创建画笔的函数是 CreatePen()，如下所示：

```
HPEN CreatePen(int fnPenStyle,//style of the pen
               int nWidth,//style of the pen
```

```
                    COLORREF crColor);//color of pen
```

nWidth 和 crColor 参数非常容易理解，但是 fnPenStyle 需要解释一下。

大多数情况下，都想画实线，但有时可能也需要画一条虚线来表示图标程序中的一些内容。可以通过画大量的被一段空格分配的实线来作为一条虚线，但是为什么不让 GDI 来做这个工作呢？线条样式支持这个功能。当 GDI 表现线条时，进行逻辑“与”运算或者掩盖住一个线条样式筛选程序。由此，就可以绘制由点和虚线、实像素或者其他任何的一维实体来构成线条。表 4.2 给出了一些可以从中选用的有效线条样式。

表 4.2 CreatePen()的线条样式

值	描述
PS_NULL	画笔不可见
PS_SOLID	画笔为实线
PS_DASH	画笔为虚线
PS_DOT	画笔为点
PS_DASHDOT	画笔为点划线
PS_DASHDOTDOT	画笔为双点划线

例如，我们创建三种画笔，每种画笔都是 1 个像素宽，样式为实线：

```
//the red pen,notice the use of the RGB macro
HPEN red_pen = CreatePen(PS_SOLID,1,RGB(255,0,0));
//the green pen, notice the use of the RGB macro
HPEN green_pen = CreatePen(PS_SOLID,1,RGB(0,255,0));
//the blue pen, notice the use of the RGB macro
HPEN blue_pen = CreatePen(PS_SOLID,1,RGB(0,0,255));
```

下面创建白色的虚线画笔：

```
HPEN white_dashed_pen = CreatePen(PS_DASHED,1,RGB(255,255,255));
```

非常简单吧！下面看一下如何向图形设备描述表中选择画笔。我们仍不知道如何绘制图形，下面首先看一下这个概念。

要将任何 GDI 对象选择到图形设备描述表，使用 SelectObject()函数，如下所示：

-
-
-
-
-

-
-

点、线、平面多边形和圆
关于文本和字体
定时的重要性
使用控件
获取信息
T3D 游戏控制程序
总结

第二部分 DirectX 和 2D 基础

第五章 DirectX 基础和令人生畏的 COM

本章中，我们将深入学习 DirectX 以及构成这项先进技术的所有基础组件。此外，我们还将详细讨论一下组件对象模型(COM)，所有的 DirectX 组件都是由 COM 构成的。如果你只是一个 C 程序员的话，那你应当十分关注这部分内容。不要担心，我们将详细地讨论该内容。

对于该部分内容，提出一点警告：在确定你不想掌握这部分内容之前，请阅读本章全部内容。DirectX 和 COM 是相互关联的，没有另外一个很难解释其中任意一部分内容。例如，如果不定义零，就无法解释零的概念。如果认为 DirectX 和 COM 的关系很简单，那你就完全错了。

下面我们要讨论的主要内容：

- DirectX 介绍
- 组件对象模型(COM)
- COM 执行的工作实例
- DirectX 和 COM 如何协调工作
- COM 的应用前景

DirectX 基础

我现在感觉自己就像是 Microsoft 的传道士，不停地将我的朋友们推向黑暗一边，但是 Microsoft 这个坏家伙一直有更好的技术！我说的对吗？

对程序员而言，DirectX 带有更多的控制功能。但实际上，这样做是值得的。基本上来讲 DirectX 是从视频、音频、输入、网络以及安装等抽象而来的软件系统，因此无论一台 PC 计算机是怎样的设置，都可以使用相同的程序。另外 DirectX 技术要比 Windows 系统自带的 GDI 和/或 MCI(媒体控制接口)速度快许多倍，功能也更强大。

图 5.1 图示了使用和不使用 DirectX 技术的情况下制作 Windows 游戏的过程。注意 DirectX 方法是多么的清晰雅致。

DirectX 的工作原理是什么呢？DirectX 提供了几乎是硬件级控制所有设备的功能。这样就可以通过组件对象模型(COM)技术和由 Microsoft 和硬件厂商编写的一套驱动程序和库函数来完成。Microsoft 提出了一套关于函数、变量、数据结构等内容的协议——硬件厂商也必须遵循以实现驱动程序与和硬件间的交流。

只要遵循这些约定，就不需要担心硬件的细节问题，只要将该内容调用到 DirectX 中，DirectX 就会完成处理细节的工作。无论视频卡、音频卡、输入设备、网卡或是其他设备是什么类型，只要是 DirectX 支持的类型，即使不熟悉这些设备，在程序中也可以使用该设备。

目前有大量的 DirectX 组件，如下所示，图 5.2 给出了其示意图。

- DirectDraw
- DirectSound
- DirectSound3D
- DirectMusic
- DirectInput

- DirectPlay
- DirectSetup
- Direct3DRW
- Direct3DIM

HEL 和 HAL

在图 5.2 中，可以看到在 DirectX 下有两个层面，分别是 HEL（硬件模拟层）和 HAL（硬件抽象层）。这是一个约定：DirectX 是一种非常有远见的设计思路，它假定可以通过硬件来实现高级性能。但是，如果硬件并不支持某些性能，那怎么办呢？这就是 HAL 和 HEL 双重模式设计思路的基础。

HAL（硬件抽象层）是接近硬件的底层。它直接和硬件交流。该层通常有设备生产商提供的设备驱动程序。可以通过常规 DirectX 调用直接和硬件进行联系。使用 HAL 要求硬件能够直接支持所要求的性能，这时使用 HAL，就能够提高性能。例如，当绘制一个位图时，使用硬件要比软件更胜任该项工作。

当硬件不支持所要求的性能时，使用 HEL（硬件仿真层）。我们以使用视频卡旋转一个位图为例来说明。如果硬件不支持旋转动作，则使用 HEL，软件算法取代硬件。很明显这样做速度要慢一些，但关键是这样做至少不会中断应用程序，应用程序仍然在运行，仅仅是速度慢一些而已。另外，HAL 和 HEL 之间的切转是透明的。如果要求 DirectX 做某工作，而 HAL 直接处理该工作，也就是硬件直接处理该项工作。否则，HEL 将调用软件仿真来处理该工作。

你可能认为软件层次太多。这是一个问题，但实际上 DirectX 非常简单，使用时你要做的可能只是一个或两个额外的函数调用。购买 2D 或 3D 图形、网络和音频加速卡花费很少，你能想像为市场上所有视频加速器来编写一个驱动程序吗？请相信我，这将需要无数的人工作无数年，也就是说这根本就做不到。DirectX 是 Microsoft 和所有的软件开发商倾注了大量努力为你提供的高性能的标准。

深入 DirectX 基础类

下面我们快速浏览一下每个 DirectX6.0 版本的组件，它们是：

DirectDraw——控制视频显示的基本着色和 2D 图形引擎，是所有图形必须穿过的通道，可能是最重要的 DirectX 组件。DirectDraw 的对象大都是系统中的视频卡。

DirectSound——DirectX 中的声音组件，只支持数字声音，不支持 MIDI。但是该组件能够使你的工作简化 100 倍，因为它处理声音时不再需要第三方的声音系统。声音编程是一种黑色艺术，过去没有人想为所有的声卡编写驱动程序。因此，许多生产商目光集中在声音效果库的市场上：Miles Sound System 和 DiamandWare Sound Toolkit。这两个系统只能允许从 DOS 系统或 Win32 平台下装载和播放数字和 MIDI 声音。但是随着 DirectSound、DirectSound3D 以及最新的 DirectMusic 组件的出现，第三方音效库已经使用得越来越少了。

DirectSound3D——DirectSound 的 3D 声音组件，使得 3D 声音在房间中环绕。这是一种相当新的技术，但是正快速地成熟着。现在大多数声卡支持硬件加速的 3D 效果，包括 Doppler 转换、折射、反射等等。但是如果使用软件仿真的话，这些都不需要了。

DirectMusic——最新的 DirectX 组件，DirectMusic 具有 DirectSound 不支持的 MIDI 技术。不只是这些，DirectMusic 具有全新的可下载声音（DLS）的系统，使得用户能够创建乐器的数字表示方式，并且能够通过 MIDI 控制来回放声音。它就像一个波形表合成器，不过是软件合成器。DirectMusic 具有各种人工智能系统的全新性能引擎。在实时运行中可以通过支持的模板改变音乐。重要的是该系统能够任意创建音乐。

DirectInput——该系统处理所有的输入设备，包括鼠标、键盘、游戏控制杆、操作杆、控制球等等。现在 DirectInput 能够支持力反馈设备，该设备具有电机传动设备和应力传感器，以允许人工显示力，因此用户能够感觉到力的存在。它正在真正地将计算机空间的“性”产业置于过载状态。

DirectPlay——这是 DirectX 的网络方面内容，允许通过 Internet、调制解调器、直接连接或任何其他能够进行交流的媒体进行抽象联系。最酷的是 DirectPlay 使得网络毫无了解的人也能建立连接，不必编写驱动程序、使用接口等等。另外，DirectPlay 支持会话的概念（正在发展中的游戏）、休息室的概念（游戏玩家聚集和玩的地方）。DirectPlay 不会强制玩家进入任何多人玩的网络结构，它只是为你传递和接收包裹。关于 DirectPlay 的内容和安全性方面已超出本书范围。

Direct3DRM——Direct3D 保留模式，它是高级的、以对象、帧为基础的 3D 系统，能够用来创建 3D 程序。它利用 3D 加速器，但速度并不是最快的。它对于编写预排程序、模型显示或速度极慢的演示是非常有用的。

Direct3DIM——Direct3D 即时模式，它是低级的 DirectX 的 3D 支持。原来由于和 OpenGL 存在许多冲突，几乎不能使用。老版本的即时模式被称为执行缓冲器，主要是数据和设备的序列，描述绘制的场景非常困难。但是，从 Direct5.0 开始，即时模式能够通过 DrawPrimitive()函数支持大量的类 OpenGL 界面。这样就可以将三角胶片和风扇等传递到着色引擎中，使用函数调用而不是执行缓冲器来改变状态。因此现在我非常喜欢 Direct3D 即时模式。尽管本卷和第二卷都是关于软件支持的 3D 游戏内容的书，为了完整起见，我们将在第二卷的最后来讨论一下 Direct3DIM。实际上，在第二卷的 CD 上有完整的关于 Direct3D 即时模式的计算机手册。

DirectSetup/Autoplay——这两种组件是准 DirectX 组件，允许一个程序从用户计算机上的应用程序安装 DirectX，当该 CD 放入系统中时立即启动该程序。DirectSetup 是一组很小的在用户计算机上装载的运行时间 DirectX 文件，并在注册表上注册这些文件。AutoPlay 是标准的 CD 子系统，在 CD 根目录上查找 AUTOPLAY.INF 文件，如果找到该文件，AutoPlay 执行该文件中的批处理命令函数。

最后，你可能疑惑 DirectX 为什么有如此多的版本。DirectX 好像每 6 个月升级一次。因为图形和游戏技术变化得太快了，所以我们从事的工作具有一定的冒险性。然而，因为 DirectX 是建立在 COM 技术基础上的，我们为之编写的程序——如在 DirectX3.0 版本上编写的程序在 DirectX7.0 版本上也能运行。下面我们看一下 DirectX 如何工作.....

COM：这是 Microsoft 的工作，还是魔鬼的？

当前编写的计算机程序很容易达到几百万行，大一点的系统几乎能达到上亿行程序代码。随着程序不断庞大，编写摘要和体系结构就具有非常重要的意义。否则，将导致全盘混乱。

C++和 Java 是计算机语言方面两种最新的尝试，即使用更多的面向对象的编程技术。C++实际上是 C 语言的演变（或者说是回流），在 C 语言中添加了面向对象技术。而 Java 则是建立在 C++基础上的，但它是完全面向对象的，并且更为简洁。另外，Java 更大程度上是一个平台，而 C++只是一种语言。

虽然使用哪种语言很重要，但归根到底，决定性的是如何用它。尽管 C++已经具有了非常酷的面向对象的技术特征，许多人还是不愿使用，或者错误地使用。因此编写大程序依然存在问题。这就是 COM 模型碰上的一个困难。

COM 很多年前作为一种新的软件范例以简单的白皮书的形式出现，这和计算机芯片或 Lego 块工作的方式相类似。只要将它们拼到一起，就可以工作了。计算机芯片和 Lego 块知道如何成为计算机芯片和 Lego 块，因此什么问题都解决了。要利用软件实现这种技术，要使用非常类似的界面，你所能想像的任何类型的函数集都具有它统一的规格。这就是 COM 做的事情。

对于计算机芯片而言，最大的好处就是当你添加更多的芯片时，不必通知其他的芯片。但是如你所知，使用软件程序来完成就困难一点。你至少要重新编译形成可执行文件，解决这一问题是 COM 的另一个目标。你应当能够向 COM 对象添加新的特征，而不会中断使用旧的 COM 对象的软件。另外，COM 对象改变后不必重新编译原来的程序就能工作，这是其最酷的技术特性。

因为能够不进行重新编译程序就可以升级 COM 对象，所以不使用修补程序或新版本就能够升级软件。例如，有这样一个程序，使用三个 COM 对象，分别实现图像、声音和网络化（见图 5.3）。现在假设该软件程序已经售出了 100000 套，但是又不希望发送 100000 套的升级版！要升级图形 COM 对象，那就给用户一个新的图形 COM 对象，程序将自动使用该图形对象。不需要重新编译，不需要连接，一切都不需要，非常简单。当然该技术使用低级语言因而非常复杂，并且需要编写自己的 COM 对象，但是使用起来则非常简单。

下一个问题是 COM 对象如何颁或包含，如何体现其即插即用的特性。答案是无章可循，但是在大多数情况下，COM 对象都 DLL。即动态连接库，随着使用的软件带来或者是下载。这样便于升级和更改。这样唯一的问题是使用该 COM 对象的软件必须知道如何从 DLL 中装载该对象。该部分内容我们将在本章“创建一个准 COM 对象”部分中讨论。

应用 DirectXCOM 对象

一个 COM 对象实际上就是实现大量界面的一个 C++类或者是一套 C++类（一个界面就是一套函数）。这些界面用于和该 COM 对象进行联系，如图可以看到一个 COM 对象，它分别具有 IGRAPHICS、ISOUND 和 IINPUT 三个界面。

每一个界面都有大量的函数可以调用（只要你知道如何使用）。因此一个 COM 对象可以拥有个或多个界面，你可以有一个或多个 COM 对象。并且，COM 技术要求说明用户创建的所有的界面必须从一个指定的基本类界面 IUnknown 中导出。对于一个 C 程序员来讲，这就意味着 IUnknown 就像是一个创建界面的字符串指针。

下面是IUnknown的类定义：

COM的前景

总结

第六章 首次接触：DirectDraw

在本章中,将第一次接触到 DirectX 的最重要的组成部分之一:DirectDraw。DirectDraw 可能是 DirectX 中最重要的技术，因为它沟通了 2D 图形的显示和 Direct3D 所依赖的帧缓冲层。只要掌握了 DirectDraw，就能够编写各种在 DOS16/32 下编写的图形应用程序。DirectDraw 是理解 DirectX 中许多概念的关键，所以要特别注意。

以下是本章的主要内容：

- DirectDraw 的界面
- 创建一个 DirectDraw 对象
- 与 Windows 协同工作
- 进入事件模式
- 巧妙的色彩
- 建立一个显示画面

DirectDraw 界面

DirectDraw 由一些界面组成。如果掌握了第五章关于 COM 的讨论，即“DirectX 基础和令人生畏的 COM”，可以知道界面只不过是一些用来和组件联系的函数和(或)方法的集合体。图 6.1 给出了 DirectDraw 界面的示意图。

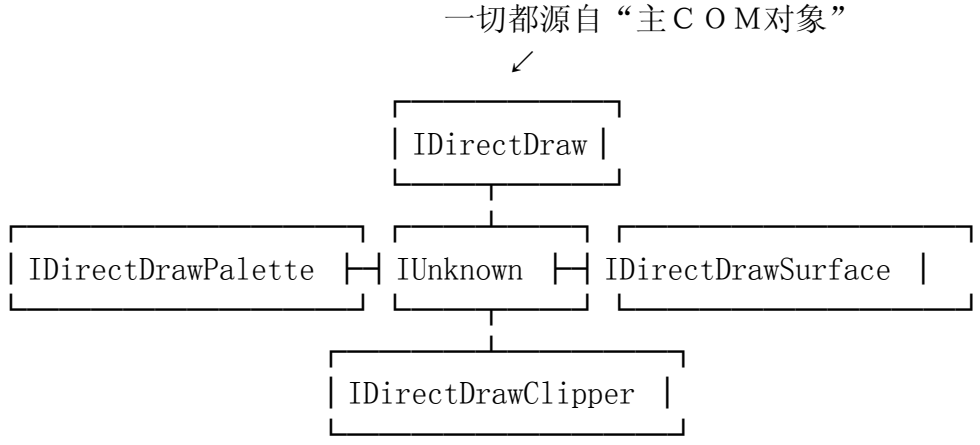

图 6.1 DirectDraw 界面

界面特征

从图中可以看到，DirectDraw 由 5 个界面组成：

IUnknown——所有的 COM 对象必须从这个基本界面中获得，DirectDraw 也不例外，IUnknown 只包含了被其他界面覆盖的 Addref()、Release()及 QueryInterface 函数。

IDirectDraw——这是一个必须创建出同 DirectDraw 一起开始工作的主要界面对象。IDirectDraw 字面上表示视频卡和支持硬件。有意思的是，现在在 MMS(多监视支持)和 Windows 98/NT 下，你可以在系统中安装多个视频卡，因此就会有多个 DirectDraw 对象。不过，本书中，即使系统中有多个卡，我们也假定在计算机中只有一个视频卡,而且总是选择一个默认卡代表 DirectDraw 对象。

IDirectDrawSurface——这代表你将创建的实际显示画面，显示时需要利用到 DirectDraw。视频卡仅利用本身的 VRAM(视频 RAM)而不用系统的存储器就能存储一个 DirectDraw 画面。

主画面通常是代表正在被视频卡光栅化并显示的实际视频缓冲，另一方面辅助画面通常不在显示区内。大多数情况下，你只需创建一个最初的画面表示实际视频显示，然后创建多个辅助画面表示对象位图和(或)用反向缓冲代表后备作图区。在此后备作图区中生成下一幅动画帧。在本章的后面部分将对画面的一些细节问题进行介绍。图 6.2 显示了一个图画的制作。

IDirectDrawPalette——DirectDraw 可以处理任何色空间，从 1 位单色到 32 位真彩色。所以，DirectDraw 支持 IDirectDrawPalette 界面处理使用 256 色或较少彩色的显示方式的调色板。这种情况下，你在一些演示中应多使用 256 色模式，因为对于软件光栅来说，256 色是最快的模式。在第二册中有关 Direct3D 即时模式的讨论中你将转而用 24 位色彩，因为 24 位色彩是 Direct3D 工作的基本方式。本例中，IDirectDrawPalette 界面用来创建、加载、操作调色板及调用调色板来画图，如主画面及辅助画面，图 6.3 表示了画面和 DirectDraw 调色板之间的联系。

IDirectDrawClipper——IDirectDrawClipper 用于一些可视显示面的子集剪切 DirectDraw 光栅和位图操作。大多数情况下，只在视窗 DirectX 应用程序软件中使用 DirectDraw 剪切器，和(或)用 DirectDraw 剪切器在显示画面的区域里对位图进行剪切。IDierectDrawClipper 界面的优点是：如果有相应的硬件，IDirectDrawClipper 界面能利用硬件加速，而且可以对在屏幕区域内通常需要剪切位图的像素及图像进行处理。

在创建一个 DirectDraw 对象前，需要重新回顾一下上一章关于 COM 的内容。DirectDraw 和所有的 DirectX 组件都处于稳步发展中，界面一直在升级。因此，即使到现在为止，本章中涉及到 DirectDraw 的界面从类属上说是指 IDirectDraw、IDirectDrawSurface、IDirectDrawPalette 及 IDirectDrawClipper，大部分界面已经更新，而且有了更新的版本。如 IDirectDraw 升级到 DirectX 第 6 版的 IDiretDraw4。

所以，如果想要得到最近的软件和硬件性能，就需要使用 IUnknown::QueryInterface()函数获得最新的修订版本。如果想弄清楚，须参看 DirectX SDK 手册。当然，本书中，使用 DirectX 6.0，所以知道哪些是最新的界面，但是，要记住，当升级到 7.0 版本的时候就会有一些你想要的更新的界面。然而，本书的两部分内容都是关于光栅转换和 3D 软件的。所以，它是很实用的。在大多数情况下，你在新版本中也几乎用不上别的技巧。

界面的协同使用

下面，将简单介绍用这些界面如何创建一个 DirectDraw 应用程序：

1.创建主 DirectDraw 对象，得到一个 IDirectDraw4 界面。在此界面上，设置协同等级和视频模式。

2.在 IDirectDrawSurface 界面上，创建至少一个主画面。基于画面的色深及视频模式。如果视频模式为每像素 8 位或更少，则需要使用调色板。

3.在 IDirectDrawPalette 界面上创建一个调色板，用 RGB 三元组初始化调色板，并把调色板附在界面上。

4.如果 DirectDraw 程序带有一个窗体，或对一个可能出了 DirectDraw 可视界面边界的位图进行着色时，你就需要创建一个剪切工具，并把它调整到可视窗体的区域中。如图 6.4 所示。

5.在主画面上画图。

当然，上述步骤省去了许多细节，但这是使用不同界面的精髓。下面我们开始认真地探讨细节问题并使这些界面真正地工作。

创建 DirectDraw 对象

用 C++创建一个 DirectDraw 对象，你需要调用 DirectDrawCreate()函数，如下：

```
HRESULT WINAPI DirectDrawCreate(GUID FAR *IpGUID,//guid of object
                LPDIRECTDRAW FAR *lplpDD,//receives interface
                Iunknow FAR *pUnkOuter);//com stuff
```

lpGuid——这是你所需要的显示驱动的 GUID(全局统一标识符)。大多数情况下，你只须用 NULL 代替默认硬件。

lplpDD——这是一个指向获得 IDirectDraw 指针的指针，注意：lplpDD 返回一个 IDirectDraw 界面，而不是 IDirectDraw4 界面。

pUnkOuter——高级特征：通常设置为 NULL。

以下是在 IDirectDraw 界面上创建一个默认 DirectDraw 对象使用的程序：

```
LPDIRECTDRAW lpdd=NULL;//storage for IDirectDraw
//create the DirectDraw object
DirectDrawCreate(NULL,&lpdd,NULL);
```

如果程序执行成功的话，lpdd 将成为一个有效的 IDirectDraw 对象界面。如果你想得到最新的界面 IDirectDraw4,得先了解一下 DirectDraw 的错误处理。

DirectDraw 的错误处理

DirectX 中的错误处理是十分清楚的。有一些能够测试出任何程序或函数执行是否成功的宏。对 DirectX 函数中的错误进行测试采用微软授权的两个宏。

FAILED()——对错误进行测试。

SUCCEEDED()——对成功进行测试。

在此基础上，应加上以下的错误处理代码：

```
if(FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
  //error
}//end if
```

同样，判断测试是否成功，你可以加上以下代码：

```
if(SUCCEEDED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
//move on to next step
}//end if
else
{
//error
```

```
}//end else
```

我通常使用 FAILED()宏，因为我不喜欢有两个不同的逻辑路径。但不管怎样，这个宏的主要问题是它本身并不能告诉你很多，它们更倾向于检测常规问题。如果你想知道确切的问题，你可以浏览一下程序返回的代码。表 6.1 列出了 DirectX 6.0 版本中 DirectDrawCreate()函数返回的代码。

表 6.1 DirectDrawCreate()返回的代码

返回代码	说明
DD_OK	成功
DDERR_DIRECTDRAWALREADYCREATED	DirectDraw 对象已经被创建
DDERR_GENERIC	DirectDraw 不知道哪出错了
DDERR_INVALIDDIRECTDRAWGUID	GUID 设备未知
DDERR_INVALIDDIRECTDRAWGUID	传递的参数有错(原文明显印错了)
DDERR_NODIRECTDRAWHW	没有任何硬件
DDERR_OUTOFMEMORY	大胆地猜猜看

常量和条件逻辑一起使用的唯一问题是，微软并不保证他们不会完全改变所有的错误代码。不过，在任何情况下你可以相当放心地使用以下代码：

```
if(DirectDrawCreate(...)!=DD_OK )
{
//error
}//end if
```

而且，所有 DirectDraw 函数定义了 DD_OK，所以你可以放心使用。

改进界面

如上所示，你可以用存储在 lpdd 中基本 IDirectDraw 界面调用 DirectDrawCreate()。或通过 IUnknown 界面的方法 QueryInterface()设法找到新的界面，把界面升级到最新版本(不管它是怎样的)。QueryInterface()是每个 DirectDraw 界面的工具。DirectX 6.0 版本的最新 DirectDraw 界面是 IDirectDraw4，所以，以下是如何获得一个界面指针。

```
LPDIRECTDRAW lpdd=NULL;//standard DirectDraw 1.0
LPDIRECTDRAW lpdd4=NULL;//DirectDraw 6.0 interface 4
//first create base IDirectDraw interface
if(FAILED(DirectDrawCreate(NULL,&lpdd,NULL)))
{
//error
}//end if
//now query for IDirectDraw4
if(FAILED(lpdd->QueryInterface(IID_IDirectDraw4,
                               (LPVOID *)&lpdd4)))
{
//error
```

```
}//end if
```

有两点需要注意：

- QueryInterface()调用的方式。
- 用来申请创建 IDirectDraw4 界面的常量是 IID_IDirectDraw4。

通常，所有的界面调用都具有以下形式：

```
interface_pointer->method(parms...);
```

所有的界面标识符用以下形式：

```
IID_IDirectCD
```

在此，C 指代组件：Draw 代表 DirectDraw，Sound 指代 DirectSound，Input 代表 DirectInput，等等。D 是个编号，从 2 到 n，表明了你想要的界面。另外，你可以在 DDRAW.H 文件中找到这些常量。

继续这个例子，你可能有点尴尬，因为你现在有一个 IDirectDraw 界面，一个 IDirectDraw4 界面。该怎样做呢？既然你不需要旧的界面，就把旧的界面释放掉，做法如下：

```
lpdd->Release();
lpdd=NULL;//set to NULL for safety
```

从现在开始，所有的方法调用都使用新的 IDirectDraw4 界面。

警告：IDirectDraw4 的新功能使一切变得井井有条，这不仅仅是因为 IDirectDraw4 界面更为先进和完美，而且在许多情况下 IDirectDraw4 界面需要并返回新的数据结构，而不是 DirectX 1.0 定义的基本结构。了解这些变化的唯一途径是查看 DirectX SDK 文献，检查具体的函数所需的或(和)返回的数据结构的版本，然而，这只是一般的警告，本书中，我将给出你要完成的所有例子的正确的结构。

除了在最初的 IDirectDraw 界面指针(lpdd)中使用 QueryInterface()函数外，还有一种可以直接得到 IDirectDraw 界面的“COM 方式”。这种方式更为直接。在 COM 下，只要你有界面标识符或称 IID（代表你想要的界面），你就可以得到任何界面的界面指针。大多数情况下，我个人不喜欢使用低水平的 COM 函数。

尽管如此，当你接触到 DirectMusic 时，除了使用低水平的 COM 函数外没有其他方法。所以，至少在此适当介绍一下这个过程。通过以下代码，你就可以直接创建一个 IDirectDraw 界面：

```
//first initialize COM,this will load the COM libraries
//if they aren't already loaded
if(FAILED(Coinitialize(NULL)))
{
//error
}//end if
//Create the DirectDraw object by using the
//CoCreateInstance() function
if(FAILED(CoCreateInstance(&CLSID_DirectDraw,
                           NULL,
                           CLSCTX_ALL,
                           &IID_IDirectDraw4,
                           &lpdd4)))
{
```

```
//error
}//end if
//now before using the DirectDraw object,it must
//be initialized using the initialize method
if(FAILED(IDirectDraw4_Initialize(lpdd4,NULL)))
{
//error
}//end if
//now that we're done with COM,uninitialize it
CoUninitialize();
```

前面的代码是微软推荐创建 DirectDraw 对象的方法。但这个技术有些不可靠，也只使用了一个宏：

```
IDirectDraw4_Initialize(lpdd4,NULL);
```

你可以去掉(1)，使用

```
lpdd4->Initialize(NULL);
```

将其代替，使之全部成为 COM。(1)(2)式中的 NULL 是视频设备，在本例中视频设备是系统设定的驱动器（这就是为什么要填为 NULL 的原因)。总之，不难明白前面的宏是怎样扩展为代码。我想可能是可以变得容易些。但是，为什么微软不继续利用宏来创建新界面，如：

```
DirectDrawCreate(...);
```

那不是更好吗？但为什么要问为什么呢？我的观点是你应当自己这样做，这样你的程序代码看起来就会相当统一。

现在，你已经知道怎样创建一个 DirectDraw 对象，以及怎样获得最新的界面。那我们就进行下一步——设置协作等级。

和 Windows 协同工作

众所周知，Windows 是一个协作、共享的环境。虽然作为一个程序员，我至今还没有想到怎样使得我的程序与 Windows 协同工作，但这至少是一种概念。无论如何，DirectX 和任何 Win32 系统相似。最低限度，DirectX 准备调用不同资源时必须通知 Windows。这样 Windows 其他的程序就不会请求调用 DirectX 所控制使用的资源。基本上，DirectX 是一个完整的资源体，Windows 知道它在做些什么。

在 DirectDraw 例子中，你唯一需要感兴趣的是视频显示硬件，下面两个模式值得注意：

- 全屏模式
- 窗体模式

在全屏模式中,DirectDraw 的运行很像旧的 DOS 程序。这就是说,DirectDraw 给你的游戏分配了全屏画面，你可以直接对视频硬件进行操作。没有其他的应用程序能够接触到硬件。窗体模式有点不一样。在窗体模式中，DirectDraw 更多的与 Windows 协作，因为其他的应用程序可能需要更新自己的客户窗体区域（对使用者是可见的）。因此，在窗体模式中，你对视频硬件的控制和独占受到了限制。然而，你仍然有足够的权限使用 2D 和 3D 加速，这是好事。但另一方面，首先会是上窄下宽...

进入事件模式

巧妙的色彩

创建一个显示画面

总结

第七章 高级 DirectDraw 和位图图形

本章将讲述 DirectDraw 的内容，并开始编写本书中所有演示程序和游戏的基础——图形库模块(T3DLIB1.CPP\H)。本章内容丰富，在本章课程中，我将给出一个图形库。尽管我们所要探讨的问题相当复杂，但我保证深入浅出地阐明这些问题。本章主要内容有：

- 真彩色模式
- 页翻转和双缓冲
- BLITTER
- 剪切
- 位图装载
- 彩色动画
- DirectX 窗口技术
- 从 DirectX 中获取信息

真彩色模式下工作
双缓冲
动态画面
页面变换
应用图形变换器
剪切基础
采用位图
备用画面
位图的放旋转和缩放
离散采样理论
色彩效果
人工色彩变换或者查询表
新的 DirectX 色彩和 Gamma 控制接口
GDI 和 DirectX 混合使用
获取 DirectDraw 的真用
在画面上冲浪
使用调色板
在窗口模式下应用 DirectDraw
总结

第八章 矢量光栅化及 2D 变换

本章讨论矢量的问题，如：如何画直线和多边形，并演示如何建立第一个 T3DLIB 游戏库。本章是数学讨论的第一章，不过，没有关系，只要花费一点点时间加上少许努力，你就会发现没有掌握不了的。在我所讲的内容中最难的部分是矩阵(仅仅是为了使您在 3D 中不至于对矩阵陌生而做的简介)。当然。根据多数人的需要，本章将就滚动、等角 3D 引擎给出一些有用的建议。记住，这是一本于 3D 的书，所以本书节省了其他方面的篇幅留给了 3D。但是，文中没有介绍的部分您会在 CD 中学到。这里列出本章的主要内容。

- 画线
- 剪切
- 矩阵
- 2D 变换
- 画多边形
- 滚动和等角 3D 引擎
- 计时器
- T3DLIB1.0

绘制线条

线框多边形
2D 平面的变换
矩阵引论
变换
缩放
旋转
填充实心多边形
多边形碰撞检测
定时与同步说解
滚动和视角场景
伪 3D 等角引擎
T3DLIB1 库函数
BOB(变换对象)引擎
总结

第九章 用 DirectInput 和力反馈进行输入

我记得有一次，我建立了一个 TTL 芯片输出操纵杆界面，使我在 Atari800 上编写的游戏能够每个针操纵杆接口可以支持四个玩家。那使我累病了吗？无论如何，输入设备已经存在很久了，DirectX 应该支持它们。本章中，我们将看看 DirectInput 以及一些输入算法，并且，我将给出一个力反馈例子。这里是你将看到的：

- DirectInput 综述
- 键盘
- 鼠标
- 操纵杆
- 输入组合
- 力反馈
- 输入函数库

输入循环回顾

DirectInput 序曲
力反馈详述
编写通用的输入系统：T3DLIB2CPP
总结

第十章 用 DriectSound 和 DirectMusic 演奏乐曲

过去在 PC 机上播放声音和音乐简直比登天还难。然而，随着 DirectSount 和 DirectMusic 的出现，这一切都变得很容易了。本章包括下面的内容：

- 发声原理
- 数码声音与合成声音
- 发声硬件
- DirectSound API
- 声音文件格式
- DirectMusic API
- 对你的库增加声音支持

PC 上的声音编程

第三部分 编程核心

第十一章 算法、数据结构、内存管理及多线程

本章将讨论在其他游戏编程参考书中疏漏的细节问题。我们的讨论内容涉及到从游戏的编写到演示的制作以及优化理论。本章将帮您掌握这些必需的编程细节。下一章，我们在讨论人工智能时，读者可以掌握一些 3D 运算，这样便对一般的游戏编程概念有了基本的了解。

本章主要有以下内容：

- 数据结构
- 算法分析
- 优化理论
- 运算技巧
- 混合语言编程
- 游戏的保存
- 多人游戏的实现
- 多线程编程技巧

数据结构

算法分析
递归
树结构
优化理论
制作演示程序
保存游戏的策略
实现多人游戏
多线程编程技术
总结

第十二章 人工智能在游戏中的运用

本章将回答许多关于人工智能的问题。实际上，人工智能不是仿真，而是基于逻辑、数学、概率、记忆的各种智能的集合——这些我们都有吗？

学习了本章内容之后，就能够编写出比较完善的程序和算法，这些程序和算法可以使游戏中的物体以合理的方式运行，而且可以做任何你想要它们做的事。本章主要内容有：

- 人工智能入门
- 单一确定的算法
- 模式和脚本
- 行为状态系统
- 存储和学习
- 计划和决策树
- 导航
- 先进的脚本语言
- 人工神经网络
- 遗传算法
- 模糊逻辑

人工智能入门

明确 AI 算法
模式和基础控制脚本
行为状态系统建模
应用软件对存储和学习建模
计划和决策树
导航
高级 AI 脚本
人工神经网络
遗传算法
模糊逻辑
在游戏中创建真正的 AI
小结

第十三章 基本物理建模

70、80 年代时，视频游戏中还没有大量包含物理学在内。大部分游戏内容为枪战、侦探-破坏游戏、探险游戏等等。然而，进入 90 年代和“3D 时代”后，物理建模就变得越来越重要了。你不能只是简简单单地使游戏中的对象按照一条不现实的路径移动，对象的移动路径至少应该大致是与现实相符。这一章包含了基本的不基于微积分的物理建模。然后，第二册覆盖更多的内容，基于微积分的 2D 和 3D 模型。以下是这一章的目录：

- 物理基本定律
- 万有引力
- 磨擦力
- 碰撞响应
- 正向运动学
- 微粒系统
- 游戏关键

大多数使用物理模型的模拟和游戏中使用的模拟都是基于标准牛顿物理学的模型。标准牛顿物理学在合理规定的尺寸和质量内（即速度比光速小得多，物体比一个单一的原子大得多，但比一个星系小得多），对于运动和物体相当适用。然而，即使建模实际用基本牛顿物理，也会占用计算机大量能量，一个简单的模拟如下雨或撞球台（如果能够正确模拟的话）将会需要用 PentiumⅢ以上处理器。

尽管如此，我们却在从 AppleⅡ到 PC 机都看见了下雨及撞球游戏。这些是怎样编写出来的呢？这些游戏的编写者了解物理，在此基础上建模，并在系统预算资源之内编程，他们创建的模型与做游戏的人在现实生活中所期待的十分接近。程序由大量的技巧、最优化、建模系统的假设和简化组成。例如：计算两个球体碰撞后的结果要比计算出两个不规则行星碰撞后的结果容易的多。所以，编程者可能会近似的把游戏中的行星用简单的球体代替(只要物理计算可行)。

在最新的游戏中，物理学需要占用很大篇幅，这是因为其中不仅仅包括物理，而且还包含需要学习的数学，所以我将只介绍一些最基本的模型。通过这些模型，在你的第一个 2D/3D 游戏中，你将可以对所需要的一切建模。我所介绍的这些物理知识并不比高中物理甚至初中物理多多少。

物理学基本定律

线性动量的物理性质：守恒和传递
万有引力效果模型
磨擦力
基本的特殊碰撞响应(高级)
解决 n-t 坐标系统
简单运动学
微粒系统
游戏关键：创建游戏的物理模型
总结

第十四章 综合运用

这一章我们打算略述一个简单游戏——*Outpost* 的设计和执行，这个游戏用到的技术都是本书中曾学习过的知识。

这个游戏用了五天时间就完成了，所以别对它期望太高。然而，它游戏的内容有 3D 模型、微粒、声效和不同的“敌人”，我认为对你来说使用它是容易的。下面是我要介绍的内容：

- Outpost 的初步设计
- 用于编程的工具
- 游戏领域：太空滚动
- 游戏者的飞船：鬼怪号
- 行星领域
- 敌人
- 能量供给
- HUD
- 微粒系统
- 执行游戏
- 编译 Outpost

Outpost 的初步设计

编制游戏的工具

游戏领域：空间滚动
游戏者的飞船“鬼怪号”
行星领域
敌人
供给能量
HUD
微粒系统
执行游戏
编译 Outpost
结束语

制作者的话

第一次做 chm 书,谢谢大家下载并阅读.

这本书在 csdn 的游戏论坛经常看到人要,不过一直没有中文版的.今天偶然得到,就把它做成 chm 文件了,方便大家共享.

目前内容是很少的,只有前两章是全的,第三章算是比较多.即使这样我相信这个版本仍然很有价值,因为这本书确实十分出色.初中级游戏编程者都能在书里得到很多东西,结构安排合理,代码写得也很好.是目前市面上能买到的最好游戏编程书了.

我希望你能通过阅读这个电子版了解这本精彩之作,并去购买正版的图书.这本书是由中国电力出版社出版的,定价 89 元.相对于其丰富的内容算是非常超值了.实际购买的话也就是 70 元左右,相信你看过这里收录的几章及目录后,更能坚定你购买的决心.大家一起来支持正版.

一项庞大的工程:
我不知道剩下的部分什么时候才能补充完成,如果你手头有这本书,希望你能帮忙录入一些,然后寄给我,下一个版本里你的名字也会出现在制作者名单里.
这部书的内容非常多,但是只要大家肯花些时间,国内所有游戏开发者硬盘里将会有一本非常有人情味的游戏编程书.
关于书中的图片,如果有可能的话截下图来,和文字排成 html 文件压缩一下寄给我.没有图片的话也没关系,只录入文字我们也非常感谢.

有新进度和新版本的话我会在 csdn 的游戏编程论坛发布

我的 email:kofight@21cn.com

Version 0.1 make by kofight

2003/7/16