VGA 实用编程技术

罗健军　编著

- 16色
- 256色
- 15位、16位高彩色
- 24位真彩色

清 华 大 学 出 版 社

(京)新登字158号

内 容 简 介

本书全面介绍了VGA的通用图形编程技术，包括最新的真彩色和高彩色模式下的编程技术，并系统介绍了与图形编程有关的字符显示、图形打印、鼠标操作、屏幕漫游及XMS操作等多项实用技术。在此基础上，本书用C++实现了一套能对VGA图形软件开发提供全面支持的程序系统。和该书配套的一张软盘以C++类库的形式提供了这套程序，同时还提供了这套程序的全部源程序及有关的一些实用程序。欲购盘者请与清华大学软件部联系。

本书适用于各类软件开发人员，特别是C++及C程序员。

图书在版编目(CIP)数据

VGA实用编程技术 / 罗健军编著. —北京:清华大学出版社,1995
ISBN 7-302-02000-0

Ⅰ.V… Ⅱ.罗… Ⅲ.计算机图形学-程序设计 Ⅳ.TP3 91.4

中国版本图书馆CIP数据核字(95)第18471号

出版者: 清华大学出版社(北京清华大学校内,邮编100084)
印刷者: 北京市海淀区清华园印刷厂
发行者: 新华书店总店北京科技发行所
开　本: 787×1092 1/16　印张: 24.25　字数: 601 千字
版　次: 1996年1月第1版　1996年1月第1次印刷
书　号: ISBN 7-302-02000-0/TP・924
印　数: 0001—5000
定　价: 26.00 元

前　　言

1. 编写本书的目的

随着计算机图形图象技术的迅速发展，特别是图形用户界面(GUI)的广泛使用，图形显示技术已越来越成为在软件开发中必须采用的一项基本技术，图形显示质量已成为决定软件开发水平的一个重要因素。

VGA是使用得最为广泛的一种微机图形显示系统，从其产生至今VGA已经历了近十年的发展历程，在这期间VGA的图形显示性能有了很大的提高，最初VGA的图形显示能力只能达到320×200×256色和640×480×16色，而目前的VGA已经具备了1600×1200×24位真彩色的图形显示能力，但由于开发工具、技术资料的缺乏及兼容性方面的问题，目前的大部分软件开发工作都还只能利用到VGA最初的图形显示能力，这无疑是对提高软件水平的一个极大制约，也是对性能优良的硬件资源的一种浪费。

本书编写的目的就是向广大程序员全面介绍通用的VGA最新实用编程技术，以使VGA的各种优良特性能在软件开发中得到广泛的采用，提高软件开发水平。

2. 本书的内容

目前微机图形图象技术应用的主要方面有：图形用户界面、CAD、三维动画、桌面排版系统、真实图象显示、游戏等，本书并不介绍这些方面的专门图形图象处理技术，本书所要介绍的是在VGA上开发这些方面的图形软件所必须掌握的基础图形显示技术及一些相关的基础技术。本书具体包含如下内容：

·VGA图形显示技术　VGA的各种显示模式下的图形操作技术以及基本的绘图技术，这是本书的主要内容。

·字符显示技术　包括点阵英文字符、矢量英文字符、点阵汉字、矢量汉字的显示技术。

·图形打印技术　EPSON系列和HP系列打印机上的图形打印技术及相应的图象缓存技术。

·鼠标驱动技术　通用鼠标操作技术及在各种扩展图形模式下的鼠标驱动技术。

·屏幕漫游技术　由VGA硬件所支持的虚屏定义、屏幕分割及屏幕移动技术。

·扩展内存操作技术　这是与图形显示无关的一项技术，因在本书的编程中多处用到扩展内存，所以将其作为本书的一部分内容。

本书分为三大部分。第一部分为第1章至第10章，主要介绍VGA的基本图形编程技术；第二部分为第11章至第14章，介绍了字符显示、图形打印、鼠标驱动及屏幕漫游技术；第三部分为第15章，介绍了本书所提供的程序系统及一些实用程序的使用方法。

3. 本书的目标

随着软件开发水平的不断提高，软件开发人员的负担也越来越重，目前一项高质量的软件开发工作往往需要同时采用与硬件及环境软件有关的多项基础技术，本书所介绍的技术仅仅只是其中的一个方面，而且所有这些技术都处于快速的发展之中，因此，软件开发人员往往没有时间来仔细研究所要用到的每一项技术，在大部分情况下他们所需要的只是

直接使用这些技术。如果本书只是介绍一些技术原理或给出一些示例性的程序，那么读者就必须对这些技术进行专门的研究并进行较繁重的二次开发工作，才能有效地使用这些技术。

为了给读者提供最直接、最有力的支持，本书将给出一套对VGA图形软件开发提供基础支持的程序系统，本书所介绍的所有技术都将完整地在这套程序系统中得到实现。通过这套程序，读者可在软件开发中直接使用本书所介绍的全部技术，而不需要再在这些技术上进行程序开发工作。

这套程序系统的作用类似于Borland C++的图形接口文件BGI和图形函数库GRAPHICS.LIB，与之相比所要实现的这套程序系统有如下优点：

·提供了对各种VGA扩展图形模式的支持，包括最新的24位真彩色模式和15位、16位高彩色图形模式；

·提供Broland C++ 10种英文矢量字符显示功能的同时，增加了点阵汉字及矢量汉字的显示功能；

·提供了图形打印、鼠标驱动、屏幕漫游功能；

·在一些大容量的数据操作中实现了对扩展内存的支持;

·提供了全部源程序，并能够修改和扩充；

·全面采用了面向对象技术，具有良好的可维护性，可方便的将其扩充到支持更多的图形模式。

4. 本书的适用情况

本书的程序全部采用C++编写。为尽量照顾使用其它语言的读者，对书中的每一项技术，在给出它们的C++程序之前，都将完整系统地介绍它们的技术原理及程序实现步骤，这些介绍是与语言无关的，这就使得各类读者都能有效地掌握本书所提供的技术，只是所能获得的支持程度不同。

C++程序员能从本书获得最直接最强的支持，本书的所有内容对这类读者都是适用的；C++程序的基本构成单位仍然是函数，而在函数层次上C++和C是基本相同的，因此本书所提供的绝大部分程序也同样适合于C语言程序员；本书中那些直接操作VGA硬件的程序都采用嵌入汇编的方式编写，这些程序对汇编语言程序员有直接的参考价值；除了具体的编程技术之外，本书的内容适合于各类软件开发人员。

在本书所介绍的图形技术方面，作者曾得到叶建平同志的大力支持和帮助,在此表示衷心感谢。

基于水平所限，加之计算机技术的迅速发展，书中错误和不当之处在所难免，恳请广大读者指正。

作 者

1995年7月

目　录

第 1 章　VGA显示系统原理

1.1　PC机图形显示系统发展概况

1.1.1　主流系统的发展

以图形显示卡的发展为标志，PC机图形显示系统的发展经历了CGA、EGA、VGA、Super VGA几个阶段，在这样一个发展过程中，PC机的图形显示性能得到了极大的改善和提高，其性能的提高主要表现在分辨率、色彩、速度几个方面。

1. CGA(Color Graphics Adapter)

这是1982年IBM在其最初推出的PC机上所采用的彩色图形适配器。CGA的工作模式分为两种：字符模式和图形模式。

在字符模式下的主要显示方式为：一屏显示80×25个字符，每个字符由两个字节来定义，第一个字节为字符的ASCII代码，第二个字节确定了字符的颜色属性，每个字符可有16种显示色、8种底色和闪烁属性。这种字符显示方式成为一种标准被一直使用至今。虽然字符最终还是以图形方式显示在屏幕上，但这时软件控制的最小单元是字符而不是象素，字符向象素的转换工作由硬件来完成，这就使得字符模式能获得比图形模式快得多的显示速度。

在图形模式下有三种不同的显示方式：640×200象素，单色，所显示的颜色可在16色中选一种；320×200象素，4色，所显示的颜色可在两组中选一组；160×100象素，16色，但BIOS驱动程序不支持这一方式，需直接操作寄存器来实现。

CGA上有16K的显示存储器，占用的主机地址为B8000H～BBFFFH。这使得CGA在80×25字符模式下可有4个显示页，在320×200模式下可有2个显示页。

CGA主要配置于8086及早期的80286微机。虽然从现在来看CGA是非常简陋的，但在图形显示方面它已完全达到了实用化的水平，在当时的CGA上已开发出了商业图形、CAD、游戏等多方面的非常丰富的图形软件，可以说微机图形图象技术的发展开始于CGA。

2. EGA(Enhanced Graphics Adapter)

1985年IBM推出了EGA。与CGA相比EGA具有更高的分辨率和更丰富的色彩，其最高图形分辨率为640×350，颜色数为64种选16种。EGA采用了与CGA完全不同的结构，但它仍然包含了CGA的所有显示模式，并且做到了在硬件上与CGA基本兼容。EGA上有256K的显示存储器，其最高显示模式可有2个显示页。EGA占用128K的主机地址空间，字符模式和图形模式各使用64K。

EGA在技术上的一个显著特点是采用了一种位面技术来组织16色模式下的显示存储器。在这种技术中，同一象素点在各位面上占同一地址，而不同位面上同一象素地址中的内容形成色彩组合。采用这一技术主要出于几方面的考虑：一是，减少地址空间的占用，解决显示存储器容量大于其所能使用的地址空间的矛盾；二是，期望在今后增加颜色数时能在显示速度及兼容性上获得好处；三是，这一技术能加快字符显示速度。但从现在来看，这一技术实际上并没有带来什么好处。目前这一技术已成为16色模式下组织显示存储器的标准方式。

EGA主要配备于286及早期的386微机，其寿命较短，而且由于它是一个基于专利技术的产品，使得兼容产品出现较晚，所以它的装机量并不是很大。

3. VGA(Video Graphics Array)

1987年IBM放弃了其所创立的PC机，推出了一种新结构的个人计算机系统PS/2，VGA即是作为PS/2的标准显示系统随PS/2一起推出的。VGA被制作在PS/2的主板上，而不是一个单独的插卡。从兼容角度考虑，IBM还单独提供了供PC机使用的VGA显示卡。

VGA的基本结构与EGA非常类似，与EGA相比它主要作了以下几方面的改进：一是，VGA采用模拟接口与显示器相连，而EGA采用的是数字接口，这一改变极大地提高了VGA的色彩显示能力；二是，EGA的绝大部分寄存器都只能写不能读，而VGA的寄存器除了少数几个外都是可读写的，这给编程提供了很大的方便；三是，EGA的显示刷新过多地占用了访问显示存储器的时间，它只给CPU留下了1/5的时间，而VGA则将其占用存储器的时间减到了最小，使得CPU能以快得多的速度来读写显示存储器。这些改进，使得VGA成为一种比较完善且极具发展潜力的产品。

VGA和EGA在硬件一级具有很好的兼容性，在VGA上可直接运行访问EGA寄存器的程序。VGA的显示存储器容量及地址占用情况都与EGA相同。

在性能上，VGA将16色模式的分辨率提高到了640×480，同时VGA新提供了一种具有320×200分辨率、256种颜色的图形模式，且所显示的每一种颜色都可从262144(18位)种颜色中选择，VGA的这种色彩显示能力对微机图形图象软件的发展起到了很大的促进作用。在16色模式下VGA仍然采用了位面技术，但在256色模式下没有采用位面技术，而是直接用连续的位来表示象素的颜色，因为这时采用位面技术就需要将存储器扩大到512K。同时256色模式的分辨率被限制在320×200，这使得它所需的存储器容量正好小于其可用的地址空间64K。

由于有仿制EGA的基础，在VGA推出的当年即有好几家厂商推出了VGA的兼容产品，并在这些产品中对VGA的性能进行了扩充。

4. Super VGA

各VGA兼容产品厂商几乎在仿制VGA的同时就开始了对VGA性能的扩展，直到目前，这一扩展工作还在继续进行，那些在性能上得到扩展的VGA兼容产品就被统称为Super VGA，而最初的VGA则被称为标准VGA。虽然与最初的VGA相比，目前的Super VGA的性能已有了很大的提高，但它们始终都没有突破标准VGA的体系结构，从技术上很难将Super VGA看作是不同于标准VGA的新一代产品，现在一般用VGA来统称标准VGA和Super VGA。

从1987年底出现第一个Super VGA产品，到现在已经历了9年时间，在这期间对VGA的性能扩展大致经历了三个阶段：

第一阶段主要是分辨率的提高，大约用了两年时间，16色及256色模式下的分辨率都被提高到了1024×768，在此之后，分辨率的进一步提高则开始受到显示器性能及显示速度的制约，在目前的普通机型上，最高分辨率仍然只能达到1024×768。

第二阶段完成了一项非常有意义的扩展，即实现了15位、16位高彩色(high color)及24位真彩色(true color)显示模式，这些图形模式使VGA具备了完全真实的色彩表现能力，并简化了某些情况下的色彩操作方式，这对多媒体、三维实体造型等图形图象技术的发展将起到十分重要的促进作用。

第三阶段主要是致力于速度的提高。速度对获得更高的分辨率及适应多媒体、三维动画等方面的应用都是一个至关重要的因素。对速度的提高主要从两方面进行：一方面，采用新的局部总线(Local Bus)接口。较早的VGA所采用的ISA总线，其数据传输率为8M/s，局部总线的数据传输率理论上可达到132M/s，局部总线的采用可极大地提高CPU读写显示存储器的速度；另一方面则是增加VGA的内部数据传输位数，最初采用的是8位，目前常见的是32位，最高则达到了128位。随着速度的加快，分辨率亦得到很大的提高，目前最高档的VGA在16色、256色模式下的分辨率可达到2048×2048，在真彩色模式下可达到1600×1200。

随着分辨率及颜色数的提高，显示存储器的容量亦不断地扩大，从最初的256K扩大到了512K、1024K，目前普及型产品都处于这一水平，而高档产品则被进一步扩大到2M、4M，目前的最高档产品达到了8M。

随着显示卡性能的增强，显示器的性能(主要是分辨率)也就需要进行相应的提高，但这方面的工作显得更加困难一些，事实上在VGA的发展过程中，显示器的发展总是落在了后面，目前普通彩色显示器的分辨率只能达到1024×768，而且大多还是采用隔行扫描，而高档显示器的分辨率则能达到1600×1200。

不能将Super VGA看作是某一种型号的产品，它实际上是一类产品，不同时期、不同档次、不同厂家的VGA产品在性能及技术结构上都存在着差异。VGA是目前PC机上唯一主要的一类显示控制器，目前还看不出什么产品有可能完全或部分地取代它。

1.1.2　其它产品的发展

这里主要介绍除上述主流产品外，在PC机上所出现过的其它一些显示控制产品，同时还将介绍PS/2所使用的几种视频显示系统

1. MDA(Monochrome Display Adapter)

这是装在IBM第一台PC机上的单色显示适配器，MDA即不能显示彩色也不能显示图形，它只能以单色方式显示字符，它一屏显示的字符数为80×25，每个字符可有正显、反显、闪烁、加亮几种属性。它只装有4K的显示存储器。在显示字符时其屏幕象素分辨率为720×348，因此其字符显示效果比CGA要好。由于MDA不具备图形显示能力，它很快就被另一种单色显示适配器所取代。

2. HGC(Hercules Graphics Card)

这是在IBM推出MDA后不久，由大力神计算机技术公司所推出的一种单色显示适配器，它具有与MDA相同的字符显示方式，但它提供了图形显示能力，这使得HGC很快赢得了市场，取代MDA成为当时单色显示的标准。HGC装有64K的显示存储器，其提供了720×348、640×400两种单色显示模式。直到VGA的单色显示系统出现后，HGC才开始逐渐被淘汰。

3. CGE400(Color Graphics Enhancer)

这是由IBM推出的一种增强型彩色图形适配器，即增强型的CGA。它推出的时间与EGA基本相同，它的性能也与EGA基本相当，但它采用了与CGA类似的体系结构，因此在技术上它与EGA是完全不同的。CGE400的显示存储器为128K，占用的主机地址空间为32K。它提供的最高图形模式为640×400×16色，在这一模式下也采用了EGA的位面技术，但地址映射却采用了与CGA类似的一种分域扫描方式。它的体系结构显然不如EGA先进、合理，而且当时人们更加关注的是EGA，因此这种产品没有得到广泛的使用。

4. MCGA(Multi-Color Graphics Array)

这是与VGA一起发布用于PS/2微机的视频系统，MCGA主要用在PS/2系列中最低档的30型机上，同VGA一样，它也是被做在PS/2的主板上。MCGA是CGA的扩充，它与CGA完全兼容。在CGA的基础上，它增加了640×480×2色，320×200×256色两种新的图形模式。由于它并不是一种很先进的产品，所以没有被移植到PC机上，而仅仅在PS/2上得到使用。

5. 8514/A

8514/A是由IBM作为PS/2微机的高分辨率图形系统随VGA一起于1987年推出的。与VGA不同，IBM没有发布可用于PC机的8514/A产品。

在IBM的产品中，8514/A可谓独树一帜，它不是从别的显示系统演变而来，也不和已有的任何显示系统兼容，这主要是由于它的独特使用方式：它插在PS/2的视频总线上与主板上已有的VGA一起工作，这时兼容性的任务可由主板上的VGA来承担。

与以往的产品相比，8514/A最本质的特点在于它具备了图形处理能力，它能完成画线、画短向量、生成坐标地址、填充矩形。在显示存储器中复制图形。向主机传送数据等多项工作。这使得它具有很快的图形显示速度。因为要将价格控制在与微机相适应的水平上，它只具备了比较简单的图形处理功能，与专业图形处理器相比还有较大的差距。

8514/A在16色和256色模式下的最高分辨率都为1024×768，但当时的显示器在正常显示方式下达不到这一水平，因此8514/A在最高分辨率下采用了隔行扫描，并采用了长余辉管显示器。

IBM一反传统，没有发表8514/A的硬件规格说明，再加上8514/A的独特使用方式及它的兼容性问题，使得8514/A的仿制和移植工作困难重重，最终还是没有出现能在PC机上使用的8514/A的兼容产品。

6. XGA(Extended Graphics Array)

1991年IBM推出了PS/2的新一代图形系统XGA，并用XGA取代VGA作为其PS/2微机的缺省图形子系统。从功能上可将XGA看作是VGA与8514/A的结合物，XGA实现了与VGA的硬件兼容，同时又具备并扩展了8514/A的图形处理能力，并在这方面保持了与8514/A的兼容。XGA的最高分辨率仍为1024×768，但其具备了16位高彩色的显示模式。在图形处理上增加了位图操作、带图案的区域填充等功能，但其仍然是一个比较简单的图形处理器。

与8514/A不同，IBM公布了XGA的硬件规格说明，但尽管如此，目前也还没有出现能用于PC机的XGA兼容产品，这可能是由于XGA与PS/2的微通道结构联系太紧密的缘故。

与Super VGA的发展相比，PS/2图形系统的发展显得更加深入而有序，而Super VGA的发展则显得更具活力。像XGA这样具有简单图形处理能力的显示系统可能是今后PC机显示系统的发展方向。

1.1.3 发展中的问题——兼容性

IBM发布了PC版的VGA之后，即全面放弃了对PC机的支持，随后各VGA兼容厂商即开始了对VGA的扩充，由此开始了Super VGA的发展历程。尽管各厂商在其Super VGA产品中都完全遵从了标准VGA的体系结构并保持了与标准VGA的兼容，但他们都不可避免地要对那些扩充功能的实现方式作出选择，而这时已失去了IBM这样一位技术上及市场上的领导者，那些Super VGA厂商就只能自搞一套，由此形成了Super VGA的混乱局面：各种型号的Super VGA在扩充

功能的使用上都有各自的编程接口，几乎没有哪两家的产品能完全相互兼容。这一兼容性问题给软件开发带来了很大的困难，使Super VGA各种扩充特性的使用受到了极大的阻碍，事实上目前除了少数大型软件及一些专用软件外，大部分软件都还只是针对标准VGA开发的，而没有用到Super VGA的扩充特性。

不久各Super VGA厂商即认识到相互兼容的重要性，于是一些主要的VGA产品供应商联合起来，组成了一个视频电子标准协会VESA(Video Electronics Standards Association)。该组织主要负责对Super VGA的编程接口及其与高分辨率显示器的接口进行标准化。1989年VESA发表了VGA BIOS扩展提案，以下称其为VESA标准，该标准统一了VGA扩展功能的编程接口，此后各厂商所设计、生产的Super VGA产品都提供了对VESA标准的支持，但它们仍然保留了自己编程接口。VESA标准的发布无疑是很有意义的，但它已经落在了产品的后面，当时各种型号的Super VGA产品在技术上和市场上都已成熟，已有了很大的装机量。虽然VESA标准马上被用到了硬件设计之中，但要使它能为软件开发所采用，则需等到支持该标准的硬件产品在装机量上占有主导地位之后，这样就需要相当长的时间。到目前，还在使用的VGA产品，不支持VESA标准的已经很少了，可以说单纯依据VESA标准来开发图形软件的时机已经完全成熟，但这毕竟已经浪费了很长的时间。

1.2　VGA显示器原理

一个完整的VGA图形显示系统由两部分组成：显示器和显示卡。显示卡负责接收和储存由主机所发出的图象数据，并对该数据进行处理和转换，生成一定的时序信号传送给显示器；显示器按照显示卡所发送的信号进行屏幕显示，最终形成与主机所发出的数据相对应的图象。

这里介绍VGA显示器的基本原理、有关概念及影响性能的因素。

1.2.1　单色阴极射线管显示器

阴极射线管(CRT)显示器一直是计算机上最主要的显示装置。最初的CRT是单色的，单色CRT的主要组成部分为：电子枪、聚焦装置、水平垂直偏转装置、荧光屏。其工作过程为：电子枪发射出电子束，电子束首先经过聚焦装置得到聚焦，然后电子束通过水平垂直偏转装置改变发射方向，最后电子束轰击到荧光屏表面的某一点上，使该点处的荧光粉发光从而在屏幕上呈现出一个亮点。电子枪发射出的电子束的强弱决定了该点的亮度，施加在偏转装置上的电压则决定了该点的位置。

电子束在一个时刻只能点亮屏幕上的一个点，但当电子束离开该点时荧光粉发出的亮光并不会马上熄灭，而是要保持一段时间。由于电子束的偏转及强弱都是由电信号的变化来实现的，它具有非常快的速度，远远超出了人的感觉能力，因此，当电子束连续快速地扫过屏幕上的若干个点时，这些点在视觉上就会被同时显示在屏幕上，构成一幅图象。

1.2.2　彩色阴极射线管显示器

彩色CRT的基本构成和原理与单色CRT是相似的，它们之间主要有如下一些不同：电子枪由一支变为三支，它们分别对应于红绿蓝(RGB)三种基色，三支电子枪发出三个电子束；偏转装置使三个电子束同时发生偏转，轰击到荧光屏上；这时荧光屏上不再是一种荧光粉，而

是间隔涂有三种荧光粉，分别对应于红绿蓝三色，为了使一个电子束只能轰击到一种颜色的荧光粉，在彩色CRT上增加了一个装置，称为荫罩。

荫罩是一个放在荧光屏之前的金属平板，上面布满了小孔，称为荫罩孔，电子束必须穿过荫罩孔才能到达荧光屏，由于三个电子束是不重合的，它们射向荫罩上同一点的入射角也就不同，当穿过某一个荫罩孔后，三个电子束就会到达荧光屏上三个分离的点上，在这三个点上分别涂上三种颜色的荧光粉，就可保证每个电子枪所发出的电子束只会点亮一种颜色的荧光粉。荧光屏上涂有不同荧光粉的点是相互分离的但又靠得非常近，当相邻的三个不同色点被点亮时，人的视觉无法分离出三个点，而只能感受到一个点的存在，并且所感受到的颜色是这三个点的颜色的综合，由此彩色CRT即可在人的视觉上形成有效的色彩组合，达到彩色显示的效果。

施加在偏转装置上的电压决定了所显示点的位置。三支电子枪所发射出的电子束的强弱则决定了一定的颜色组合，而各电子束的强弱可以连续平滑地调整，因此彩色CRT能够显示出任何一种颜色，它的色彩显示能力是不受自身限制的。

VGA显示系统所采用的即是CRT显示器，一般是彩色CRT，但也可采用单色CRT。

1.2.3 光栅扫描

电子束扫过一幅屏幕图象上的各个点的过程称为屏幕扫描，屏幕扫描有两种不同的方式：随机扫描和光栅扫描。在随机扫描方式下，根据所要显示的每个图形元素(通常是直线或点)的坐标值来控制电子束的偏转，电子束只扫过屏幕上那些需要显示的部分。在光栅扫描方式下，电子束按固定的路径扫过整个屏幕，在扫描过程中，通过电子束的通断强弱来控制电子束所经过的每个点是否显示或显示的颜色。

在早期的计算机显示器中曾采用过随机扫描，现在的显示器采用的都是光栅扫描。

光栅扫描的路径通常为：从上到下扫过每一行，在每一行内从左到右扫描。其扫描过程如下：

电子束从屏幕的左上角开始向右扫，当到达屏幕的右边缘时，电子束关闭(**水平消隐**)，并快速返回屏幕左边缘(**水平回扫**)，又在下一条扫描线上开始新的一次水平扫描。一旦所有水平扫描均告完成，电子束在屏幕的右下角结束并关闭(**垂直消隐**)，然后迅速返回到屏幕左上角(**垂直回扫**)，开始下一次光栅扫描。

1.2.4 分辨率

在光栅扫描显示器上存在一个分辨率的概念，分辨率指屏幕上所能显示的象素点的个数，它一般用水平象素数×垂直象素数的形式来表示，如640×480。垂直象素数等于光栅扫描中水平扫描线的条数。水平象素数等于一条水平扫描线中所能显示的点的个数，它实际上等于一次水平扫描期间，电子束的通断强弱状态能够发生变化的次数。

早期的显示器只能在某个固定的分辨率及能被该固定分辨率偶除的分辨率下工作。现在的显示器一般都能在多种分辨率下工作。显示器所能达到的最高分辨率是直接衡量显示器性能的一个主要指标，目前的彩色显示器在最高分辨率上大致有三个档次：普通产品1024×768、中档产品1280×1024、高档产品1600×1200。下面所要讨论的一些因素对显示器的分辨率都有着直接影响。

1.2.5　扫描频率

·垂直扫描频率　荧光粉所发出的亮光只能保持一定的时间，为了使图象能持续、稳定地保留在屏幕上，电子束就需要以一定的频率不断地重新扫描整个屏幕，这一过程称为**屏幕刷新**。垂直扫描频率即是指每秒钟的屏幕刷新的次数，其也称**帧频**、**场频**。帧频对画面质量有着直接的影响，过低的帧频会使屏幕图象出现明显的闪烁，提高帧频则能减少闪烁并增加屏幕亮度。目前的显示器的帧频大都在50Hz～90Hz。提高帧频本身并不存在困难，但它要受到行频、点频及分辨率的限制。

·水平扫描频率　指电子束每秒钟所扫过的水平扫描线的数量，也称**行频**。在帧频一定时，行频决定了一屏所能具有的水平扫描线的数量，显示器所能达到的最高行频是决定显示器分辨率的主要因素。目前，普通显示器的行频一般在30kHZ～38kHz，高档显示器的行频则可达到30kHz～85kHz。

·带宽　指在1秒钟内电子束的通断强弱状态能发生变化的最大次数，也称**点频**。在行频一定时，点频决定了一条扫描线内所能显示的象素点的数量，点频也是决定显示器分辨率的一个主要因素。目前，普通显示器的点频一般在45MHz左右，最高档的显示器则达到了200MHz。

早期的显示器具有固定的帧频、行频和点频，因此它们只能显示一种分辨率。而目前的显示器大都可以在一个较大的范围内调整自己的工作频率，以适应不同分辨率的需要。

在帧频一定时，可以计算出在某一分辨率下所需行频和点频的最低限。如帧频为60Hz，分辨率为640×480时，可算出每秒需要显示的扫描线为28800条，需要显示的象素点为18432000个，则对应的行频为28.8kHz，点频为18.432MHz。实际所需的频率比计算值要高，以补偿回扫所占用的时间，对行频大约需补偿10%，对点频约需补偿30%。

1.2.6　隔行扫描(interlaced)

按上面的计算方式，可估计出帧频为60Hz时，1024×768分辨率所需的行频和点频分别为50kHz和62MHz，这已超出了大多数普通显示器行频和点频的最高值，但这些显示器仍然能够显示1024×768的分辨率，这就是因为它们采用了隔行扫描技术。所谓隔行扫描就是在屏幕刷新中电子束每隔一行作一次水平扫描，在两次屏幕刷新中完成对一屏所有扫描线的扫描。隔行扫描使每次屏幕刷新所扫描的线数减少了一半，因此可以具有比较小的行频和点频。与隔行扫描相对应，正常的扫描方式就被称为**逐行扫描**(non-interlaced)。

由于隔行扫描会产生比较明显的屏幕闪烁，在采用隔行扫描方式时帧频一般都被提高到80Hz以上，这时可基本消除闪烁现象。

在大多数普通显示器上，800×600是用逐行扫描方式所能实现的最高分辨率，但这时其行频已经接近了最高限，因此其通常具有较低的帧频，一般为56Hz，虽然与通常所采用的60Hz帧频相差无几，但这时会明显感觉到闪烁现象，其显示效果反而不如采用隔行扫描具有较高帧频的1024×768分辨率。

在一定的技术水平下，行频和点频的提高都会带来成本的增加，因此可以说隔行扫描是在性能和价格之间进行折中的一种很好方式。

1.2.7 屏幕尺寸

显示器屏幕的大小通常用屏幕对角线的长度来表示，其单位一般为英寸，1英寸等于2.54cm。通用显示器屏幕的宽高比总是为4:3，因此屏幕宽度等于对角线长度的4/5，高度等于对角线长度的3/5。由于在屏幕边缘会出现较严重的图象失真，所以图象一般不会占满整个屏幕，实际使用的屏幕的宽度和高度一般为满屏的0.9。

目前PC机上标准显示器的尺寸为14英寸，也有少数为15英寸。超过17英寸的显示器即被称为大屏幕显示器，目前大屏幕显示器的常见尺寸有：17英寸、20英寸、21英寸、29英寸、33英寸。

增大屏幕尺寸是进一步提高显示器分辨率的一个基本前提，事实上1024×768基本已是14英寸显示器分辨率的极限，只有增大屏幕尺寸，进一步提高分辨率才会得到满意的效果，目前具有较高分辨率的高档显示器大都为大屏幕显示器。

将17英寸的显示器作为个人计算机的标准显示器会得到极好的显示效果，但目前在价格上还不现实，目前17英寸显示器的价格为普通14英寸显示器的5～7倍，相当于一台微机的价格。

1.2.8 点距

点距指荫罩上水平方向上相邻的两个荫罩孔中心点之间的距离。目前显示器的点距一般为0.28mm或0.31mm，最小的达到了0.24mm，较差的为0.39mm，这给出的只是一个平均值，一般在屏幕中央点距较小，在屏幕边缘则较大，对标定点距为0.28mm的显示器，其屏幕中央的点距一般为0.27mm，屏幕边缘则为0.30mm。

点距是影响高分辨率显示效果的一个重要因素，点距越小所显示的图象就越清晰。

根据屏幕尺寸及点距可以大致算出一条水平扫描线所对应的荫罩孔的数量。对14英寸的显示器，点距为0.28mm时，其每条扫描线所对应的荫罩孔的数量约为914个；点距为0.31mm时，该数量约为826个，这都要小于显示器的最高水平分辨率1024，但在该分辨率下这两种点距的显示器都能得到比较清晰的图象，因此不能认为点距直接决定着分辨率，显示器的最高分辨率可以大于荫罩所提供的分辨率，但它们不能相差太大，1024:826可以作为确定这两者适当差别的一个参考值。

1.2.9 荧光粉余辉

荧光粉余辉指荧光粉被电子束点亮后其发光所持续的时间。增长荧光粉的余辉也可以消除屏幕闪烁现象，在屏幕闪烁问题上余辉和帧频是两个互补的因素，当余辉较长时就可以允许有较低的帧频，当余辉较短时就需要有较高的帧频，增长荧光粉的余辉并没有什么困难，但较长的余辉会产生图象变化时的模糊重影现象。

早期的高分辨率显示器一般都采用了长余辉荧光粉，以弥补帧频的不足，在这些显示器上，当一行字符向上滚动时可明显看到字符在原处留下的余辉，当在原处又显示上新的字符时，这些字符就会存在一段模糊期。目前的显示器大都将荧光粉的余辉控制在不会产生重影的范围内，这时帧频应保持在60Hz以上，才能避免闪烁现象。

1.3　VGA 显示卡原理

1.3.1　VGA的结构

VGA的主要组成部分及各部分的基本功能如下。

1. 显示存储器(Display Memory)

VGA的显示存储器是一种动态随机存储器(DRAM或VRAM)，显示存储器以一定的数据格式完整地保存着当前所要显示的一整屏图象，主机的CPU通过向显示存储器写入数据来实现图象显示，显示卡则根据存储器中所存储的数据来控制显示器进行图象显示。

2. 图形控制器(Graphics Controller)

图形控制器位于CPU和显示存储器之间的数据通道上，在CPU读写显示存储器时，图形控制器是透明的。图形控制器提供了对写入存储器的数据进行逻辑操作、移位操作，对读出数据进行比较的功能，通过对图形控制器编程即可获得对这些功能的硬件支持，加快图形操作速度。

3. CRT控制器(CRT Controller)

CRT控制器产生时序信号，以对显示器的光栅扫描及屏幕刷新进行同步控制，它控制着显示器的分辨率、各种扫描频率及扫描过程的时序，同时它还控制着光标状态、屏幕与显示存储器的映射关系。

CRT控制器的时序状态一般是通过BIOS来设置，直接编程来修改它的时序状态是比较困难的。通过对CRT控制器编程，可改变屏幕与显示存储器的映射关系，由此可用硬件实现滚动、漫游、屏幕分割、多显示页等特殊功能。

4. 数据串行发生器(Serializer)

数据串行发生器从显示存储器中读取图象数据，并将所读出的数据串行化，发送给属性控制器或数模转换器。

5. 属性控制器(Attribute Controller)

属性控制器主要在16色模式下起作用，它有一个16色对256色的颜色查找表，通过该表将4位的颜色值转换为8位的颜色值，然后将这8位的颜色值传送给数模转换器。在256色模式下属性控制器将被忽略。

VGA的属性控制器继承于EGA，在EGA上其颜色查找表为16色对64。在VGA上属性控制器实际上并无存在的必要。

6. 数模转换器(DAC)

数模转换器有一个8位对18位的颜色查找表，它将来自属性控制器或数据串行发生器的8位颜色值转换为18位的颜色值，这18位颜色值被分成3个6位，分别用来表示红绿蓝三色的亮度，最后，这三个6位值被转换为三个模拟信号值发送给显示器，以控制三个电子束的强弱。

在真彩色和高彩色模式下，红绿蓝三色的亮度值直接来自数据串行发生器，这时颜色查找表将不起作用。

通过编程可修改属性控制器和数模转换器中的色彩查找表，从而可以在一个很宽的范围内对16色和256色模式下的颜色值进行设置。

7. 定序器(Sequencer)

定序器产生点和字符的时钟，控制整个适配器上所有功能的时序。定序器还用于确定字符发生器的位置，仲裁CPU及适配器对显示存储器的访问，屏蔽CPU对某一位面的修改。

1.3.2 VGA显示模式

VGA在任一时刻都必须工作在某一显示模式之下，它的显示模式分为字符模式和图形模式两大类。

字符模式也称文本模式，在字符模式下，最基本的操作单元为字符，一屏所能显示字符的行数和列数及字符属性的表示方法即构成了一种具体的字符显示模式。VGA标准的字符显示模式为80列、25行、16色。字符模式具有极快的显示速度，但它不能显示图形。

在图形模式下，最基本的操作单元为单个象素点，一定的象素分辨率及一定的色彩表示方式即构成了一种图形显示模式。VGA的图形模式分为三类：CGA、EGA兼容图形模式；标准VGA图形模式；VGA扩展图形模式。后两种模式统称为VGA图形模式。本书将只介绍VGA图形模式的编程技术，而不涉及对字符模式和CGA、EGA兼容图形模式的介绍。

VGA可以支持多种分辨率及多种色彩表示方式，多种分辨率与多种色彩表示方式相组合即形成了非常丰富的VGA图形显示模式。

1.3.3 色彩表示方式

VGA显示器及显示器与显示卡的接口都采用模拟方式来处理色彩，因此它们都具有无限的色彩显示和传输能力。但主机和显示卡只能用数字方式来表示和处理色彩，在用数字方式表示色彩时，如果要获得更丰富更细腻的色彩就需要增加表示色彩的数据位数，这就需要更大容量的显示存储器，相应的也就需要有更高的处理速度，而同时分辨率的提高也对显示存储器的容量提出了很高的要求。为了尽量降低对显示存储器容量的需求，在VGA上采用了一种间接色彩表示方式：用一个索引值来确定各个象素点的颜色，而不是直接用红绿蓝三基色的亮度值来确定每个象素点的颜色，然后用一个色彩查找表来确定每个索引值所对应的真实颜色值。索引值占用较少的数据位，而用较长的数据位来定义色彩查找表中的真实颜色值，这样既能减少显示存储器的容量，又能获得丰富细腻的色彩显示能力，但这时在同一屏图象中所显示的不同颜色的数量要受到索引值取值范围的限制。

VGA的颜色查找表采用18位数据来定义一个真实颜色，红绿蓝三种基色各占6位，每种基色可有64级亮度，总共可组合出262 144(256K)种颜色。VGA的颜色索引值采用了4位和8位两种数据长度：4位数据可有16种不同的取值，其对应于VGA的16色模式，这时一屏只能显示16种不同的颜色；8位数据可有256种不同的取值，其对应于VGA的256色模式，这时一屏可显示256种不同的颜色。这种采用间接色彩表示方式的图形模式称为**间接色彩模式**，较早的VGA都只具有这种间接色彩模式。

随着显示速度的提高、存储器价格的下降及应用要求的提高，在VGA上又出现了直接用红绿蓝三基色的亮度值来确定每个象素点颜色的**直接色彩模式**。在直接色彩模式中用16位或24位来定义一个象素的颜色，16位的模式称为**高彩色模式**，24位的模式称为**真彩色模式**。在高彩色模式中，每个基色占5位或6位数据、有32级或64级亮度，总共有32 768(32K)种或65536(64K)种颜色；在真彩色模式中，每个基色占8位数据、有256级亮度，总共有

16777216(16M)种颜色。与间接色彩模式相比，直接色彩模式完全消除了同屏颜色数的限制，简化了软件的色彩操作方式，并可以获得更加细腻的色彩。

24位的直接色彩模式之所以被称为真彩色模式，是因为这种模式事实上已达到了CRT显示器色彩表现能力的极限，在这种模式下人的视觉已无法从屏幕上分别出相邻两种颜色的差别，这时进一步增大表示色彩的位数已不再有意义。在某些计算机图形系统中存在着一种32位的色彩表示方式，但在这种表示方式中，用来描述颜色值的仍然只有24位，剩下的8位被用来描述三维图象处理中所需的一些属性。

表1-1列出了VGA所支持的各种色彩模式及它们的有关特性。在某些VGA上还有4色和2色模式，但这些模式很少被使用，且不被VESA标准支持。

表1-1　VGA色彩模式

色彩模式	数据位	同屏颜色数	最大颜色数	亮度级	表示方式
16色	4	16	262144(256K)	64	间接色彩
256色	8	256	262144(256K)	64	间接色彩
15位高彩色	15	不限	32768(32K)	32	直接色彩
16位高彩色	16	不限	65536(64K)	32或64	直接色彩
24位真彩色	24	不限	16777216(16M)	256	直接色彩

对直接色彩模式有时也用数据位数或最大颜色数来表示，如15位高彩色有时也被称为15位色或32K色，24位真彩色有时被称为24位色或16M色。

1.3.4　分辨率

显示器所显示的分辨率由显示卡来控制，VGA可以在一个很大的范围内支持多种分辨率，表1-2列出了VGA所支持的各种分辨率及有关特性。

表1-2　VGA的分辨率

分辨率	整屏象素数	宽高比	水平修正率	垂直修正率
320×200	64000	1.6	1.2	0.8333
512×480	245760	1.067	0.8	1.25
640×400	256000	1.6	1.2	0.8333
640×480	307200	1.333	1	1
800×600	480000	1.333	1	1
1024×768	786432	1.333	1	1
1280×1024	1310720	1.25	0.9375	1.0667
1600×1200	1920000	1.333	1	1
2048×2048	4194304	1	0.75	1.3333

在选择分辨率时，分辨率的宽高比是一个需要考虑的因素，它指水平象素数与垂直象素数之比。在软件中通常是以象素为单位对图形的尺寸进行控制，这时如果分辨率的宽高比与

屏幕的宽高比1.333不一致，所显示的图形就会发生变形，如圆变成椭圆，正方形变成长方形。水平修正率是在分辨率的宽高比与屏幕不一致时，用来对图形的水平尺寸进行变换的系数，它等于分辨率宽高比除以屏幕宽高比。垂直修正率是水平修正率的倒数，它用来对图形的垂直尺寸进行变换，对一幅图形只需从一个方向进行修正。对图形的修正总会降低软件的作图速度，并且会使软件变得复杂，因此在软件中应尽量采用那些宽高比与屏幕一致的分辨率，即水平修正率和垂直修正率都为1的分辨率。

表2-2所列出的只是VGA上常见的一些分辨率，在某些VGA上还提供了其它一些不常用的分辨率。事实上通过直接对VGA的CRT控制器编程，就可以在显示存储器容量及扫描频率所允许的范围内对VGA的分辨率进行任意设置，但这项工作比较复杂而且无法保证兼容性。

1.3.5 图形模式

上述各种色彩模式与各种分辨率相组合即可得到VGA的各种图形模式，但这种组合是不完全的，在某些色彩模式下不具有某些分辨率。一种图形模式一般表示为：水平象素数×垂直象素数×色彩模式，如800×600×256色。

一定的分辨率对应有一定的象素数，一定的色彩模式则决定了每个象素在显示存储器中所占用的位数，因此每一种显示模式都有一定的存储器需要量。一种模式所需的存储器容量可按下式计算：

所需存储器容量＝水平象素数×垂直象素数×每象素所占容量

如640×480×64K色模式所需的存储器容量为：

640×480×16Bit＝640×480×2Byte＝614400Byte＝600K

15位高彩色模式下，每个象素点使用15位数据，但占用了16位，其每象素所占容量与16位色模式相同，也为2个字节。由于存储器分页方面的原因，24位真彩色模式所需的存储器容量比按上述方式所计算出的容量要大。其它模式下存储器需要量都可按上述方式直接计算。

另一方面，VGA的显示存储器容量并不能任意配置，它通常必须为256K乘2的n次方，即只能为如下几种容量之一：256K、512K、1MB、2MB、4MB、8MB。因此显示卡所配置的存储器容量一般要大于其所能支持的各种显示模式的需要量，如要支持上述640×480×64K色模式，显示卡需配置1M的存储器。

从目前的价格来看，所需配置的存储器容量超过1M或分辨率超过1024×768的显示模式都属于高级模式，要想获得对这些模式的支持，硬件价格就会有大幅度的增长，目前的绝大多数PC机都没有提供对这些高级模式的支持。那些非高级模式则称为普通模式，为VESA标准所支持的普通模式共有18种，本书主要介绍这些普通图形模式下的编程技术。

表1-3列出了VGA的各种常见图形模式及有关特性。

在表1-3所列的这些图形模式中，只有640×480×16色和320×200×256色这两种模式为标准VGA的图形模式，其它都为VGA的扩展图形模式。

由表1-3可得到各种容量的显示卡在各种色彩模式下所能支持的最大分辨率如表1-4所示，表中列出的仅仅是一种可能性，某些显示卡在直接色彩模式下往往不能支持到其存储器容量所允许的最大分辨率。

表1-3　VGA图形模式

图形模式	所需容量	最低配置容量	类　　型
640×　480×　16色	150K	256K	普通模式
800×　600×　16色	235K	256K	普通模式
1024×　768×　16色	384K	512K	普通模式
1280×1024×　16色	640K	1M	高级模式
1600×1200×　16色	938K	1M	高级模式
2048×2048×　16色	2048K	2M	高级模式
320×　200×256色	63K	256K	普通模式
640×　400×256色	250K	256K	普通模式
640×　480×256色	300K	512K	普通模式
800×　600×256色	469K	512K	普通模式
1024×　768×256色	768K	1M	普通模式
1280×1024×256色	1280K	2M	高级模式
1600×1200×256色	1875K	2M	高级模式
2048×2048×256色	4096K	4M	高级模式
320×　200×32K色	125K	256K	普通模式
512×　480×32K色	480K	256K	普通模式
640×　480×32K色	600K	1M	普通模式
800×　600×32K色	938K	1M	普通模式
1024×　768×32K色	1536K	2M	高级模式
1280×1024×32K色	2560K	4M	高级模式
1600×1200×32K色	3750K	4M	高级模式
320×　200×64K色	125K	256K	普通模式
512×　480×64K色	480K	256K	普通模式
640×　480×64K色	600K	1M	普通模式
800×　600×64K色	938K	1M	普通模式
1024×　768×64K色	1536K	2M	高级模式
1280×1024×64K色	2560K	4M	高级模式
1600×1200×64K色	3750K	4M	高级模式
320×　200×16M色	200K	256K	普通模式
640×　480×16M色	960K	1M	普通模式
800×　600×16M色	1407K	2M	高级模式
1024×　768×16M色	2304K	4M	高级模式
1280×1024×16M色	3840K	8M	高级模式
1600×1200×16M色	5625K	8M	高级模式

表1-4 各种容量的显示卡所能支持的最高分辨率

显示卡容量	16色	256色	32K、64K色	16M色
256K	800×600	640×400	320×200	320×200
512K	1024×768	800×600	512×480	320×200
1M	1600×1200	1024×768	800×600	640×480
2M	2048×2048	1600×1200	1024×768	800×600
4M	2048×2048	2048×2048	1600×1200	1024×768
8M	2048×2048	2048×2048	1600×1200	1600×1200

1.3.6 速度

速度是任何一个计算机图形显示系统都必须考虑的一个重要因素。VGA系统的图形显示速度主要取决于两方面，一是主机系统的速度，二是显示卡本身的速度。

在主机系统方面，影响速度的一个主要因素是CPU的速度，最早的VGA用于286系统，现在的VGA则用在Pentium系统中，这期间CPU的速度提高了几十倍，但整个系统的图形显示速度并没有提高这么多，这主要是因为其它方面的速度没有得到同样幅度的提高。

CPU要通过系统总线向显示存储器传送数据，系统总线的速度对图形显示速度有着重要的影响。较早的PC机长期采用的是ISA总线，其数据传输率只有8M/s，ISA总线一直是图形显示的瓶颈。随后出现了一种EISA总线，其数据传输率为33M/s，但因其价格太高没有得到广泛使用。目前486和Pentium系统大都采用了局部总线，VESA和Inter分别推出了一种局部总线标准：VESA局部总线(VL Bus)和PCI局部总线，这两种局部总线的工作频率为33MHz，数据宽度为32位，理论上它们的数据传输率为132M/s，实际数据传输率VESA可达到107M/s，PCI则为76M/s，虽然后者较慢，但它似乎具有更好的发展前景，极有可能成为未来的主流。局部总线的采用基本消除了图形显示的总线瓶颈。在其它条件相同时，VESA局部总线大约能使图形显示速度提高2到4倍。

显示卡的速度取决于显示卡的工作频率及显示存储器与显示卡之间的内部数据传输位数。由于显示卡的数据处理过程必须与显示器的光栅扫描及屏幕刷新保持同步，其工作频率的提高就受到限制，因此提高显示卡速度的主要方法就是增加其内部数据传输位数，最早的VGA为8位，随后提高到16位，支持ISA总线的VGA最高为32位，采用局部总线接口的VGA则至少为32位，少数高档产品达到了64位，最新的一种支持PCI的高档VGA产品达到了128位，其所能达到的图形显示速度为64位VGA的1.6倍，为32位VGA的9倍。目前有一种方法被用来提高VGA的工作频率，即在VGA上安装一个缓存器，在缓存器之前VGA采用更高的频率工作，在缓存器之后则采用与显示器相匹配的频率工作，这种方法主要用在一些高档的产品中。

显示卡的速度会产生两方面的影响，一方面，当显示卡从显示存储器读取数据时，它将禁止CPU访问显示存储器，如果提高显示卡的速度就能给CPU留下更多的时间，使得CPU能以更快的速度将数据写入显示存储器；另一方面，显示卡必须以一定的帧频反复读取、转换、传送显示存储器中的数据，显示卡在一个刷新周期内所能处理的数据量的大小，决定了显示卡所能支持的最大显示存储器容量，从而也就决定了显示卡所能支持的最高分辨率和最高色

彩数。目前，ISA总线的16位和32位显示卡最大只能支持1M的显示存储器，32位的局部总线卡可支持2M，64位卡可支持到4M，最新的128位卡则可支持8M。

1.3.7　编程接口

对VGA的编程需要结合使用以下三种方式来完成：视频BIOS功能调用、访问显示存储器、寄存器操作。

视频BIOS是以中断调用的形式提供的一组软件接口，它对标准VGA的各种功能提供了较全面的支持，但它的图形操作速度太慢且没有提供对VGA扩展特性的支持，因此不能完全依赖此软件接口进行编程，而必须直接与VGA的硬件打交道才能实现有效的图形操作。

要想利用VGA的扩展特性并获得满意的速度，直接访问VGA的显示存储器是实现图形显示的唯一方法，正确访问显示存储器的关键是要搞清VGA显示存储器的结构，而在不同的显示模式下显示存储器都具有不同的组织结构。

在某些显示模式下，访问显示存储器之前必须对VGA的有关状态进行设置，这就需要对VGA的寄存器进行操作。通过寄存器操作还可用硬件实现一些特殊的显示功能。

下一章将详细介绍VGA的编程接口。

第 2 章　VGA图形操作技术

本章将全面介绍那些直接与VGA打交道的图形操作技术，但不涉及具体的编程。所作的介绍针对于VGA的体系结构，而并不是针对于各项图形功能，如画点、读点、画线等。编程中某项图形功能的实现往往要用到多方面的操作技术，如何联合使用多项操作技术来有效地实现一项图形功能，这将在以后的各章中结合具体的编程来介绍。

2.1　显示存储器结构

VGA在其显示存储器与屏幕图象之间建立了一种自动的对应关系，软件只需向显示存储器写入图象数据，即可实现图象显示，而将数据正确写入显示存储器的关键就是要掌握显示存储器的结构。

2.1.1　如何读写显示存储器

VGA的显示存储器被映射到主机系统的一段内存地址空间上，这段内存地址称为视频地址，在图形模式下视频地址空间为：A0000H～AFFFFH，这段内存地址在1M以下，因此不论是工作在实模式或是保护模式下的程序都能对这段内存地址进行访问，访问的方法与常规的内存操作完全一样。当向这段视频地址写入数据时，所写的数据即被写到显示存储器上，当从这段地址读数据时，所读出的数据即来自于显示存储器。

2.1.2　位面技术及存储器分页

显示存储器所能使用的地址空间只有64K，而显示存储器的容量有256K至8M，那么如何通过这段较小的地址空间来访问整个显示存储器呢？VGA采用了两种解决方法：位面技术和显示存储器分页。

1. 位面技术

在位面技术中，VGA将多段大小为64K的显示存储器同时映射到一个64K的视频地址空间上，每段64K的显示存储器即被称为一个位面，当向视频地址写入某个数据时，该数据会被同时写到所有位面上，但是可通过设置VGA的图形控制器及定序器的某些状态，来禁止或允许数据被写到某些位面，并可使所写入的数据只对整个字节中的某一位有效。由此通过一个64K的地址空间即可访问多个64K的显示存储器。

位面技术继承于EGA，它只用在VGA的16色模式中，在其它色彩模式中使用此技术会降低显示存储器的利用率，因此其它色彩模式没有采用位面技术而都采用了存储器分页的方法。

2. 存储器分页

在这种方式下，全部显示存储器被分成若干个64K的段，在某一时刻只有一段显示存储器被映射到视频地址空间上，此时即可对该段显示存储器进行访问，通过某一操作可以改变映射到视频地址空间上的显示存储器段，由此即可实现对所有显示存储器的访问。

每一个64K的显示存储器段称为一页，每页有一个页号，其对应于页在显示存储器中的排列位置，从0开始记数，即第1页的页号为0。改变映射到视频地址空间的显示存储器页的

操作，称为换页操作。当前映射到视频地址空间的显示存储器页，称为显示存储器的当前页。

在16色模式中，一个象素占有4位，因此设置有4个位面，这时能同时访问的显示存储器容量为256K，它最大能支持800×600的分辨率，当分辨率进一步提高时，又会出现地址空间上的矛盾，这时就需在位面技术的基础上再采用存储器分页方法，在这种情况下每个存储器页的容量为256K而不是通常的64K，但在一个位面上仍为64K。

存储器分页是一种更直接、更简便且效率更高的方法，但在标准VGA上只有640×480×16色模式存在地址空间的矛盾，而它采用了位面技术来解决，因此在标准VGA上没有提供存储器分页方法，这一方法是在Super VGA上产生的，所以各种型号VGA的分页操作方法都是不一样的，VESA标准提供了分页操作的软件接口，但在直接进行硬件操作时仍然没有标准。

2.1.3　地址计算

1. 地址映射关系

在所有的VGA图形模式下，显示存储器与屏幕象素点之间的映射关系都是线性的，即显示存储器中的每个象素数据按光栅扫描的顺序依次对应于屏幕上的每个象素点。在使用位面技术时，象素在每个位面上各占一个数据位，每位具有相同的地址；在没有采用位面技术时，一个象素点的多个数据位在显示存储器中连续存放，只是在24位色模式下有些特殊。下面描述屏幕上的某一个象素点在显示存储器中所处位置的计算方法。

2. 变量说明

屏幕坐标原点为屏幕左上角，向右为X轴的正方向，向下为Y轴的正方向。

X，Y象素点在屏幕上的位置；

Width——当前分辨率下的水平象素数；

Height——当前分辨率下的垂直象素数；

BitN——当前色彩模式下每个象素所在显示存储器中所占的二进制位数；

PlaneN——当前色彩模式下所具有的位面数，在没有使用位面技术时，设该数为1；

BitPP——在当前色彩模式下每个象素点在地址空间中所占的二进制位数；

ScanLeng——一条扫描线在地址空间中所占用的字节数；

Page——象素点在显示存储器中所处的页；

Offset——象素点在所处页中的地址偏移量；

Bit——象素点在所在字节中所处的位(仅在16色模式有用)。

另有两个在编程中需要考虑的参数：

TotalPage——在当前显示模式下的总页数；

SmSL——地址空间尺寸(64K)与ScanLeng相除的余数，当该数等于零时意味着可以整除，该数大于零则意味着不能整除。

上面两个参数可以反映出显示存储器分页的情况，当TotalPage＝1时，表示不分页，否则就需要分页，分页又分成两种情况，当SmSL＝0时，表示分页只会出现在两条扫描线之间，即行外分页，否则分页就会出现在一条扫描线上，即行内分页。显示存储器的分页情况对具体编程会产生影响。

3. 地址计算方法

BitPP＝BitN/PlaneN

ScanLeng＝Width*BitPP/8 (不适于24位色模式)

Page＝(ScanLeng*Y+X*BitPP/8)/10000H

Offset＝(ScanLeng*Y+X*BitPP/8)%10000H

Bit＝X%8 (只在16色模式下需要)

在24位色模式下，如果每个象素点连续存放在显示存储器中，这时分页就必然会出现在某个象素点上，为避免这种情况，在每条扫描线之后空出了若干字节。在320×200分辨率下，每条扫描线占用1024个字节，其实际只需要960个字节；在640×480分辨率下，每条扫描线占用2048个字节，其实际需要1920个字节。在每条扫描线中，各象素点数据靠前连续存放。在24位色模式下ScanLeng不能通过计算得到，而需要直接取得。

另两个参数的计算方法如下：

TotalPage＝(ScanLeng*Height-1)/10000H+1

SmSL＝10000H%ScanLeng

4. 地址参数

表2-1列出了各种普通图形模式下的有关地址参数

表2-1 各种VGA图形模式的地址参数

图形模式	BitN	PlaneN	BitPP	ScanLeng	ToatlPage	分页情况
640×480×16色	4	4	1	80	1	不分页
800×600×16色	4	4	1	100	1	不分页
1024×768×16色	4	4	1	128	2	行外分页
320×200×256色	8	1	8	320	1	不分页
640×400×256色	8	1	8	640	4	行内分页
640×480×256色	8	1	8	640	5	行内分页
800×600×256色	8	1	8	800	8	行内分页
1024×768×256色	8	1	8	1024	12	行外分页
320×200×32K色	16	1	16	640	2	行内分页
512×480×32K色	16	1	16	1024	8	行外分页
640×480×32K色	16	1	16	1280	10	行内分页
800×600×32K色	16	1	16	1600	15	行内分页
320×200×64K色	16	1	16	640	2	行内分页
512×480×64K色	16	1	16	1024	8	行外分页
640×480×64K色	16	1	16	1280	10	行内分页
800×600×64K色	16	1	16	1600	15	行内分页
320×200×16M色	24	1	24	1024	4	行外分页
640×480×16M色	24	1	24	2048	15	行外分页

2.1.4　数据格式

在显示存储器中的每个象素点的数据格式随色彩模式的不同而不同，下面分别介绍。

1. 16色模式

在这种色彩模式下，每个象素使用4位数据，这4位数据分别位于处在同一地址的4个位面上。 4个位面的编号为0、1、2、3，它与颜色索引值的4个数据位直接对应，即颜色索引值的位0存于位面0，位1存于位面1，照此类推。

在该模式下，显示存储器中存储的是每个象素点颜色的索引值，4位数据对应有16个不同的颜色索引值，通过两个色彩查找表可确定出每个颜色索引值所对应的真实颜色值。在系统初始化时，这16个索引值对应着一组默认的真实颜色值，如果在程序中不打算修改色彩查找表，就可认为这4位数据定义的是每个象素点的真实颜色值，这时每个数据位的颜色含义如下：位0——蓝色；位1——绿色；位2——红色；位3——加亮。这4种状态相互组合即可形成16种真实颜色。

在这种模式下，必须辅以寄存器操作才能有效地读写显示存储器。

2. 256色模式

每个象素使用8位数据，1个字节，每个字节连续存放在显示存储器中，这种数据存放方法被称为压缩象素法。

这也是一种间接色彩模式，1个字节对应着256个颜色索引值。在默认状态下，也可认为这一个字节定义着每个象素的真实颜色值，但这时该字节的每个位不存在有规律的基色对应规则。

3. 24位真彩色模式

每个象素使用24位数据，共3个字节。这是一种直接色彩模式，24位数据直接定义着象素点的真实颜色值。其最高8位数据，即第1个字节，确定蓝色的亮度；中间8位，即第2个字节，确定绿色的亮度；最低8位，即第3个字节，确定红色的亮度。每个基色亮度值的取值范围为：0～255，0为最暗，255为最亮。在这里，三基色的存放顺序与通常的习惯相反。

4. 15位高彩色模式

每个象素占用16个数据位即2个字节，但它只使用了15位，最高1位没有使用。在其使用的低15位数据中：位10～位14，定义蓝色的亮度；位5～位9，定义绿色的亮度；位0～位4，定义红色的亮度。每种基色亮度值的取值范围为：0～31，31为最亮。

5. 16位高彩色模式

每个象素使用16位数据，2个字节。其中：位11～位15，共5位，定义蓝色的亮度；位5～位10，共6位，定义绿色的亮度；位0～位4，共5位，定义红色的亮度。蓝、红两基色亮度值的取值范围为：0～31，31为最亮。绿色亮度值的取值范围为：0～63，63为最亮。

2.2　VGA寄存器

访问16色模式下的显示存储器、修改色彩查找表及实现一些特殊的显示功能都需要对VGA的寄存器进行操作。这里将列出标准VGA的所有寄存器，但只详细介绍那些在编程中会用到的那些寄存器。

2.2.1 概述

对VGA寄存器的操作通过I/O指令来实现，在汇编语言中用out和in两条指令，在C语言中使用inportb()和outportb()两个函数。

标准VGA包含60多个寄存器，为了避免占用过多的主机I/O地址，VGA的大部分寄存器都采用了一种间接访问方式：将寄存器分为若干组，每组使用两个I/O端口，一个为索引端口，一个为数据端口。每组中的每个寄存器都有一个索引值，通过向索引端口写入一个索引值，来选择一个所要访问的寄存器，然后通过数据端口对所选的寄存器进行读写。在各组寄存器中数据端口的地址都为索引端口的地址加1。对VGA寄存器的操作一般使用8位的I/O指令，但对这种间接访问方式也可采用16位的I/O指令，即将索引值放在AL中，数据值放在AH中，然后将AX写入索引端口地址，一次完成对那些具有间接访问方式的寄存器的写操作，这种方式速度较快，也较简洁，但某些VGA不支持这种方式。

VGA寄存器分为六组：即外部寄存器、CRT控制器寄存器、定序器寄存器、图形控制器寄存器、属性控制器寄存器和数模转换器寄存器。其中外部寄存器和数模转换器寄存器没有采用间接访问方式，而属性寄存器则还采用了一种更特殊的访问方式。

表2-2列出了VGA在彩色模式下I/O地址的使用情况，在单色模式下有所不同。

表2-2 VGA彩色模式下的I/O地址

I/O地址	寄 存 器
3CCh	混合输出寄存器(只读)
3C2h	混合输出寄存器(只写)
	输入状态寄存器0(只读)
3DAh	特征控制寄存器(只写)
	输入状态寄存器1(只读)
3C3h(46E8h)	VGA允许寄存器
3D4h，3D5h	CRT控制器寄存器
3C4h，3C5h	定序器寄存器
3CEh，3CFh	图形控制器寄存器
3C0h	属性控制器寄存器
3C6h，3C7h，3C8h，3C9h	数模转换器寄存器

以下将分组介绍标准VGA的各个寄存器。各种型号的Super VGA在标准VGA的基础上又增加了一些自己特有的寄存器，尽管这些扩展的寄存器有时也是很有用的，但由于缺乏标准，通常都要避免使用它们，因此这里不介绍这些扩展的寄存器。

2.2.2 外部寄存器

这组寄存器在编程中很少使用。

1. 混合输出寄存器

端口地址：写入时为3C2h，读出时为3CCh。

位： 0 CRT控制寄存器I/O地址选择

1 使能显示存储器

2—3　设置时钟频率(点频)
4　禁止视频输出
5　奇/偶页
6　水平回扫极性
7　垂直回扫极性

2. 特征控制寄存器

端口地址：彩色3DAh，单色3BAh，只写。

位：　0　特征控制位0
1　特征控制位1
2—3　保留
4—7　未用

该寄存器只在EGA上用，VGA保留所有7位的值，且位3必须设为0。

3. 输入状态寄存器0

端口地址：3C2h，只读。

位：0—3　未用
4　开关检测
5　特征控制位0(只EGA用)
6　特征控制位1(只EGA用)
7　垂直回扫中断(IRQ2)

4. 输入状态寄存器1

端口地址：彩色3DAh，单色3BAh，只读。

位：　0　显示允许
显示器水平扫描期间此位为1，水平或垂直回扫期间此位为0。
1　光笔选通(仅EGA用)
2　光笔开关(仅EGA用)
3　垂直回扫
垂直回扫期间此位为1。
4—5　诊断(EGA用)
6—7　未用

5. VGA允许寄存器

端口地址：3C3h或/和46E8h。

位：　0　VGA允许/禁止
允许(1)和禁止(0)对VGA存储器和寄存器(除本寄存器外)进行读和写操作。
1—7　保留

2.2.3　CRT控制器寄存器

该组共有25个寄存器。其中几个寄存器可用来实现屏幕滚动、屏幕漫游、屏幕分割等特殊功能。当要设置一种不被BIOS所支持的分辨率时，必须详细了解该组寄存器，但通常没有必要进行这项工作。

该组寄存器使用两个I/O地址，在彩色模式下索引端口地址为3D4h，数据端口地址为3D5h。在单色模式下，两个端口地址分别为3B4h和3B5h。每个寄存器有一个索引值。

1. 水平总数(索引值0)

以字符数计的一次水平扫描全过程(包括两端过扫及水平回扫)被划分成的点数，点对应于一个时间单位。

2. 水平显示允许(索引值1)

正常显示区的点数。

3. 开始水平消隐(索引值2)

水平消隐开始的点数。

4. 结束水平消隐(索引值3)

位：0—4 结束消隐

结束水平消隐的点数。

5—6 偏移控制

显示开始的延迟点数。

7 VGA芯片测试，始终为1。

5. 开始水平回扫(索引值4)

水平回扫开始的点数。

6. 结束水平回扫(索引值5)

位：0—4 结束水平回扫

结束水平回扫的点数。

5—6 水平回扫延迟

7 结束水平消隐位5

以上6个寄存器确定了水平扫描的时序及水平分辨率，当要自己设置分辨率时，需要使用这些寄存器。

7. 垂直总数(索引值6)

以字符数计的一次刷新周期内总的水平扫描线数(包括过扫及回扫)。

8. 溢出寄存器(索引值7)

存放其它寄存器中第8位以上的值。

位： 0 垂直总数位8

1 垂直显示允许结束位8

2 垂直回扫开始位8

3 开始垂直消隐位8

4 行比较位8

5 垂直总数位9

6 垂直显示允许结束位9

7 垂直回扫开始位9

9. 预置行扫描行(索引值8)

位：0—4　预置行扫描

垂直回扫结束后的起始象素行号值。用于文本方式下的象素级垂直滚动。

5—6　字节平移控制

7　未用

10. 最大扫描线(索引值9)

位：0—4　最大扫描线

每个文本字符的扫描线数(字符高度)。

5　开始垂直消隐位9

6　行比较位9

7　二次扫描

11. 光标开始(索引值0Ah)

位：0—4　光标开始

在一个字符上光标开始的扫描线位置。

5　光标关闭

6—7　保留

12. 光标结束(索引值0Bh)

位：0—4　光标结束

在一个字符上光标结束的扫描线位置。

5—6　光标偏移

7　保留

13. 开始地址高位(索引值0Ch)

14. 开始地址低位(索引值0Dh)

开始地址可用来改变屏幕与显示存储器的对应关系，从而实现垂直滚动、水平滚动、屏幕漫游、多显示页等功能。开始地址确定了显示存储器的哪一个字节位置对应于屏幕上的第一个扫描点。在不同的模式下一个单位的开始地址可能对应着2、4或8个象素点。当进行屏幕漫游、水平滚动时，该寄存器必须与偏移量寄存器(索引值13h)结合使用。

15. 光标位置高位(索引值0Eh)

16. 光标位置低位(索引值0Fh)

17. 垂直回扫开始(索引值10h)

18. 垂直回扫结束(索引值11h)

位:0—3　垂直回扫结束

4　清除垂直中断

5　允许垂直中断

6　刷新周期选择

7　索引值为0～7的CRT控制寄存器写保护

19. 垂直显示允许结束(索引值12h)

20. 偏移量(索引值13h)

它确定了一条扫描线在显示存储器中所占的字节数，该字节数可大于一条屏幕扫描线实际所需的字节数，从而可在显示存储器中定义一个比实际屏幕要宽的虚拟屏幕。该寄存器与开始位置寄存器(索引值0Ch及0Dh)结合使用可实现屏幕水平滚动及屏幕漫游。在不同的模式下，一个单位的偏移量可能对应为4、8或16个象素。

21. 下划线位置寄存器(索引值14h)

仅用于单色文本模式。

22. 开始垂直消隐(索引值15h)

23. 结束垂直消隐(索引值16h)

垂直总数、垂直回扫开始、垂直回扫结束、垂直显示允许结束、开始垂直消隐、结束垂直消隐这些寄存器及相应的溢出位，确定了垂直扫描的整个时序及垂直分辨率，当要自己设置分辨率时，需要使用这些寄存器。

24. 模式控制寄存器(索引值17h)

位： 0 CGA兼容模式支持
1 选择行扫描寄存器
2 垂直记数乘2
3 隔次记数
4 输出控制(仅EGA用)
5 地址回绕
6 字模式或字节模式
7 垂直和水平回扫复位

25. 行比较(索引值18h)

在VGA扫描过程中维持着一个行计数器，记录着当前一次刷新的已扫描行数，当行计数器等于该寄存器的值时，显示转到显示存储器地址0。当该寄存器的值小于屏幕总扫描线数时，即可实现屏幕分割，这时屏幕的上半部分显示的是开始地址寄存器所指向的数据，屏幕下半部分则始终显示存储器中位置0开始的数据。屏幕上半部分可以用开始地址寄存器来进行滚动和漫游，而同时屏幕下半部分保持静止。

行比较是一个10位寄存器，其位8放在溢出寄存器中，位9放在最大扫描线寄存器中。

2.2.4 定序器寄存器

定序器控制着所有VGA操作的总体时序，它还完成一些显示存储器的控制。它包含5个寄存器，通过两个I/O地址来访问，其索引端口地址为3C4h，其数据端口地址为3C5h。其映象屏蔽寄存器是与16色模式下显示存储器操作密切相关的一个寄存器。

1. 重置寄存器(索引值0)

位： 0 异步重置
1 同步重置
2—7 保留

2. 时钟模式寄存器(索引值1)

位： 0 8/9点字符
1 频宽(仅EGA用)

2　移位装载
3　点时钟二分频
4　32位锁存
5　关闭显示
此位为1，停止显示刷新，屏幕全黑，使CPU对显示存储器的访问最大化。
6—7　保留

3. 映象屏蔽寄存器(索引值2)

位：　0　位面0写允许
1　位面1写允许
2　位面2写允许
3　位面3写允许
4—7　未用

该寄存器用于在16色模式下，允许和禁止写入视频地址空间的数据被写到某一或某些位面上，1为允许，0为禁止。该寄存器对16色模式下的所有写方式(见图形控制器模式寄存器)产生影响。

4. 字符发生器选择寄存器(索引值3)

位：0—1　字符发生器选择A
2—3　字符发生器选择B
4　字符发生器选择B高位
5　字符发生器选择A高位
6—7　保留

5. 存储模式寄存器(索引值4)

位：　0　字符模式(仅EGA用)
1　扩展存储器(超过64K)
2　奇/偶模式
3　链接4
4—7　未用

2.2.5　图形控制器寄存器

这是与16色模式下显示存储器操作密切相关的一组寄存器，它控制着对显示存储器每个位面及一个字节中每一位的访问，并对数据的逻辑运算提供了硬件支持。它包含9个寄存器，其索引端口地址为3CEh，数据端口地址为3CFh。

1. 设置重置寄存器(索引值0)

位：　0　写入位面0的数据
1　写入位面1的数据
2　写入位面2的数据
3　写入位面3的数据

4—7 保留

在写方式0(见模式寄存器位0、位1)下，当某个位面为设置重置允许(见设置重置允许寄存器)时，在CPU写显示存储器时，将用该寄存器相应位面的数据填充到显示存储器的相应位面中，而在该位面上忽略CPU所写入的数据，那些未被设置重置允许的寄存器仍然接收CPU写入的数据。

在写方式3下，将用该寄存器的值与位屏蔽寄存器的值相与，写入到每个位面中，这时完全忽略CPU所写入的数据，并且不需要设置重置允许。

设置重置寄存器可用来以某种预先定义的颜色向显示存储器作快速填充。

2. 设置重置允许寄存器(索引值1)

位： 0 位面0设置重置允许
1 位面1设置重置允许
2 位面2设置重置允许
3 位面3设置重置允许
4—7 保留

当某个位面所对应的值为1时，在写方式0下，将用设置重置寄存器相应位面的值来写入显示存储器，而在该位面上忽略CPU所写的数据。

3. 颜色比较寄存器(索引值2)

位：0—3 4位颜色值
4—7 未用

在读方式1(见模式寄存器位3)下，用该寄存器的值与所读的象素点的4位颜色值相比较，若相等，返回给CPU的数据的相应位置1，否则置0。比较结果要受颜色忽略寄存器的影响。

4. 数据循环移位寄存器(索引值3)

位：0—2 向右循环移动的位数
3—4 逻辑功能选择
00b 直接写入数据
01b 写入数据与锁存器内容AND
10b 写入数据与锁存器内容OR
11b 写入数据与锁存器内容XOR
5—7 保留

在VGA上，对应于16色模式下的每一个位面，存在4个8位的锁存器，锁存器中存放着CPU最近一次读操作所读出的各个位面的数据。

循环移位对写方式0和写方式3有效，逻辑功能对写方式0和写方式2有效。这两个功能都只对来自CPU的数据产生影响。如果同时进行两种操作，那么循环移位将在逻辑运算之前进行。

5. 读位面选择寄存器(索引值4)

位：0—1 要读出的位面(0～3)
确定读方式0所要读的位面。
2—7 保留

6. 模式寄存器(索引值5)

位：0—1　写方式(0、1、2、3)

2　保留

3　读方式(0、1)

4　奇/偶模式(用于字符模式及仿真CGA)

5　移位寄存器方式(仿真CGA)

6　256种颜色模式

象素在每个位面中占2位，形成256色模式，而非通常的256色模式。

7　保留

在16色模式下，VGA提供了4种写显示存储器的方式和两种读显示存储器的方式：

写方式0　定序器中的映象屏蔽寄存器使写入的数据对各位面有效或无效；位屏蔽寄存器使写入的1个字节的各位有效或无效；设置重置允许寄存器确定了那些有效位面及有效位的数据是来自于CPU还是来自于设置重置寄存器；当循环移位寄存器被置位时，来自CPU的数据将执行循环移位或逻辑运算。这是一种最灵活的写方式，但在很多情况下它的速度不如其它方式。

写方式1　将锁存器的内容写入显示存储器，来自CPU的数据被完全忽略。除映象屏蔽寄存器外其它的寄存器都对该方式没有影响。这种方式主要用于屏幕图象拷贝。

写方式2　写入数据的低4位被分别写到4个位面中，且同时对地址字节的8位发生作用，位屏蔽寄存器可使写入的数据对地址字节的各位有效或无效，映象屏蔽寄存器可使写入的数据对各位面有效或无效。对一般的写点操作，这是一种最快的写方式。

写方式3　写入的数据经循环移位后与位屏蔽寄存器的值相与，结果作为位屏蔽码，设置重置寄存器的值再与该屏蔽码相与写入显示存储器，这时无条件地使用设置重置，各位面不需要设置重置允许。这种方式主要用于区域填充。在EGA上无此方式。

读方式0　一次读出一个位面的一个字节，即连续8个象素点在一个位面中的值。通过读位面选择寄存器确定所要读的位面。读一个象素点需进行4次读操作。通常的读点都采用此方式实现。

读方式1　将所读象素的颜色值与颜色比较寄存器的值相比较，若相等，返回数据的相应位置1，否则置0。这时CPU所读出的不是象素点的颜色值，而是地址字节中的8个象素点的颜色是否与颜色比较寄存器相同的标志。颜色比较的结果受颜色忽略寄存器的影响。用这种方式读一个象素点的颜色值平均需进行8次操作，所以通常情况下不采用这种读方式。这种方式主要用于任意区域填充时，确定区域边界。

7. 混合寄存器(索引值6)

位：　0　图形方式

1表示VGA处于图形显示模式，0表示处于字符显示模式。

1　奇/偶允许(用于字符模式)

2—3　存储器映射

确定视频地址空间的位置和大小

00b　A0000h/128K

01b　A0000h/64K

10b B0000h/32K
11b b8000h/32K
4—7 未用

8. 颜色忽略寄存器(索引值7)

位： 0 忽略位面0
1 忽略位面1
2 忽略位面2
3 忽略位面3
4—7 保留

在读方式1进行颜色比较时，该寄存器可使4位颜色值的某些位在比较中被忽略，置0的位面将被忽略。

9. 位屏蔽寄存器(索引值8)

当CPU按字节寻址写数据时，所写入的数据会同时对1个字节中的8位生效，即一次写操作会影响到8个象素点，该寄存器可对那些不希望被改变的位进行屏蔽。当该寄存器的某位置1时，地址字节中相应位就将接受所写入的数据，当置0时，则不接受所写入的数据。

但实际上VGA并没有进行真正的屏蔽，它只是用锁存器中的数据来填充那些被屏蔽了的位。因此要想得到正确的屏蔽结果，就必须在写操作之前对同一地址进行一次读操作。

2.2.6 属性控制器寄存器

属性控制器中存在一个色彩查找表，其称为调色板，它用于将16色模式下的4位颜色索引值转换成8位的颜色索引值，然后输出到VGA的数模转换器，象素的真实颜色值最终将由数模转换器来确定。在16色以上的色彩模式中，属性控制器不起作用。

属性控制器包含20个数据寄存器，它只使用了一个I/O地址3C0h，写入操作在索引寄存器和数据寄存器之间自动交替进行。可通过向I/O地址3DAh进行一次读写操作来进行初始化，初始化之后，对端口3C0h的第一次写操作将指向索引寄存器。

1. 索引寄存器

位：0—4 数据寄存器索引值
5 调色板地址源
0 调色板可以修改，清除屏幕
1 调色板不能修改
6—7 未用

2. 调色板寄存器(共 16 个，索引值 00h～0Fh)

位：0—5 颜色值
6—7 保留

在EGA上，低6位的颜色值被作为最终的颜色输出到显示器。在VGA上，则用颜色选择寄存器的位2、位3补足其高两位后输出到数模转换器。这16个调色板寄存器维持着一个16对64(或256)的色彩查找表，属性控制器还以另外一种方式提供了一个色彩查找表。对调色板寄存器的设置必须在垂直回扫期间进行。

3. 模式控制寄存器(索引值10h)

位：　0　图形/字符方式
　　1　单色/彩色
　　2　允许线形图形字符
　　3　设置背景亮度
　　4　保留
　　5　水平移动兼容
　　6　象素宽度
　　7　P4、P5源选择

P4、P5源选择置0时，采用16个调色板寄存器所维持的色彩查找表。该位置1时，将忽略调色板寄存器，而将颜色选择寄存器的4位颜色值作为高位，补到象素颜色值上形成8位的颜色索引值输出到数模转换器，由此提供了另一个色彩查找表。

4. 过扫描颜色寄存器(索引值11h)

位：0—7　8位颜色索引值

为了保持屏幕对中，VGA水平和垂直扫描的范围都大于屏幕图象的宽度和高度，超出的部分称为过扫描。过扫描会在正常的屏幕图象周围形成一个边框，该边框通常为黑色，所以不被察觉，此寄存器可以改变该边框的颜色。

5. 彩色位面允许寄存器(索引值12h)

位：　0　允许输出位面0
　　1　允许输出位面1
　　2　允许输出位面2
　　3　允许输出位面3
　　4—5　视频状态多路转换(EGA用)
　　6—7　保留

该寄存器可在屏幕图象输出时屏蔽某些位面的数据。

6. 水平移动寄存器(索引值13h)

位：0—3　水平移动象素数
　　4—7　保留

该寄存器可使屏幕图象以单个象素为单位进行水平移动，但其只能向左移动，且最多只能移动8个象素。该寄存器器可以作为对CRT控制器中开始位置寄存器的补充，以实现更细致的水平移动。

7. 颜色选择寄存器(索引值14h)

位：　0　颜色4
　　1　颜色5
　　2　颜色6
　　3　颜色7
　　4—7　保留

颜色6和颜色7用作输出到DAC的8位颜色索引值的高2位。在调色板寄存器被忽略时，颜色4和颜色5用作8位颜色索引值的位4和位5。

2.2.7 数模转换器寄存器

数模转换器中存在一个色彩查找表，它用于将8位的颜色索引值转换为18位的真实颜色值，DAC还负责将真实颜色值转换为三个模拟电信号输出到显示器。通过DAC寄存器可以修改其颜色查找表，从而实现对间接色彩模式下象素的颜色的设置和修改，通过这项操作可简便地实现淡入淡出、动画等一些特殊的显示效果。

1. DAC状态寄存器(只读寄存器)

端口地址：3C7h

位：0—1 DAC状态

11b DAC处于写状态，这时CPU可对查找表数据寄存器进行写操作

00b DAC处于读状态，这时CPU可对查找表数据寄存器进行读操作

2—7 保留

2. 查找表读索引寄存器(只写寄存器)

端口地址：3C7h

位：0—7 要读的查找表数据寄存器索引值(0～255)

向该寄存器进行一次写操作后，DAC即处于读状态，这时可连续对所选的查找表数据寄存器进行三次读操作，读出的三个数据分别为红绿蓝三色的亮度值。

3. 查找表写索引寄存器

端口地址：3C8h

位：0—7 要写的查找表数据寄存器索引值(0～255)

向该寄存器进行一次写操作后，DAC即处于写状态，这时可对I/O地址3C9h进行三次写操作，依次将一个真实颜色值的红绿蓝三基色的亮度值写入所选的查找表数据寄存器。

4. 查找表数据寄存器

端口地址：3C9h

共有256个查找表数据寄存器，它们与8位颜色索引值的256个取值状态一一对应。通过向查找表读索引寄存器或查找表写索引寄存器写入一个8位的索引值，即可对相应的某个查找表数据寄存器进行读或写操作。对一个查找表数据寄存器的读或写必须连续进行3次，依次对应于红绿蓝三基色的亮度值，每次读或写的数据仅低6位有效。当对某个查找表数据寄存器进行完3次操作后，将自动转到对下一个查找表数据寄存器进行操作，而不必再次设置索引值。

2.3 视频BIOS

2.3.1 概述

BIOS是对VGA进行操作的软件接口，使用此软件接口编程在兼容性上能得到最大的保证，对某些功能它提供了唯一标准的实现方法。但BIOS的图形操作速度太慢且对VGA扩展特性的支持很弱，因此也不能完全依赖它进行编程。

BIOS是固化在硬件上的一组程序，它以中断调用的方式供应用程序使用，其中断向量号为10H。在汇编语言中用软件中断指令INT来实现对它的调用，在C语言中用int86()、int86x()、geninterrupt()等函数来实现对它的调用。

目前VGA上的BIOS功能可分为三类，一是，标准VGA的BIOS功能，它为所有型号的VGA所支持；二是，VESA发布的扩展VGA BIOS功能，它提供了实现VGA扩展特性的标准方法，被目前的绝大部分VGA所支持；三是，各种型号的VGA自己特有的一些BIOS功能，它们没有统一的标准，因此在编程中应避免使用。

下面将分别介绍标准VGA的BIOS及VESA的扩展VGA BIOS，此书将只介绍那些与图形操作有关的功能，同时列出主要用于字符模式的功能。

2.3.2　标准VGA BIOS

功能0：模式选择

输入参数：AH＝0

AL＝模式号

如果AL的位7等于0，将清除显示存储器，等于1将不改动显示存储器。

返回值：无

这个功能用来设置VGA的显示模式，这是在进行屏幕输出前首先要完成的一项操作。当用该功能来设置VGA的各种扩展显示模式时，这时各种扩展显示模式没有统一的模式号，因此无法保证兼容性。对VGA的各种扩展显示模式的设置，一般应采用VESA扩展BIOS的模式设置功能来完成，这样才能保证程序的兼容性。

功能1：设置光标尺寸

功能2：设置光标位置

功能3：读光标尺寸和位置

功能4：取光笔(仅EGA用)

功能5：选择当前显示页

功能6：文本窗口向上滚动

功能7：文本窗口向下滚动

功能8：读光标所在位置的字符和属性

功能9：在光标所在位置写字符和属性

功能0Ah：在光标所在位置只写字符

功能0Bh：设置CGA调色板

功能0Ch：写图形象素

输入参数：AH＝0Ch

AL＝象素的颜色值

CX＝象素列数

DX＝象素行数

如果AL的位7设置为1，那么象素的颜色值将与当前点的已有颜色值进行异或运算。

返回值：无

该功能一般仅对640×480×16色这一种图形模式有效。

功能0Dh：读图形象素

输入参数：AH＝0Dh

CX＝象素列数

DX＝象素行数

返回值：AL＝象素颜色值

该功能一般仅对640×480×16色这一种图形模式有效。

功能0Eh：写字符且推进光标

功能0Fh：取当前显示模式

功能10h：设置EGA调色板寄存器

其包括16个对颜色进行控制的子功能，它们对应于属性控制器寄存器及DAC寄存器的各项功能。

(1)子功能0：设置调色板寄存器

输入参数：AH＝10h

AL＝0

BL＝调色板寄存器号(0～0Fh)

BH＝颜色数据(0～3Fh)

返回值：无

(2)子功能1：设置边框颜色(过扫描)

输入参数：AH＝10h

AL＝1

BH＝颜色数据(0～0FFh)

返回值：无

(3)子功能2：设置所有调色板寄存器

输入参数：AH＝10h

AL＝2

ES:DS＝17字节的缓冲区地址(16个调色板值和1个过扫描值)

返回值：无

(4)子功能3：背景亮度/闪烁控制

(5)子功能7：读单个调色板寄存器

输入参数：AH＝10h

AL＝7

BL＝调色板寄存器号(0～15)

返回值：BH＝调色板寄存器值

(6)子功能8：读边框颜色寄存器

输入参数：AH＝10h

AL＝8

返回值：BH＝边框颜色寄存器值

(7)子功能9：读所有调色板寄存器

输入参数：AH＝10h

AL＝9
ES:DX＝17字节的缓冲区地址
返回值：存储在[ES:DX]中的17个字节

(8)子功能10h：设置单个DAC寄存器
输入参数：AH＝10h
AL＝10h
BX＝DAC寄存器号(0～255)
DH＝红色亮度值(0～63)
CH＝绿色亮度值(0～63)
CL＝蓝色亮度值(0～63)
返回值：无

(9)子功能12h：设置DAC寄存器组
输入参数：AH＝10h
AL＝12h
BX＝起始DAC寄存器(0～255)
CX＝要设置的寄存器数目(1～256)
ES:DX＝颜色表的地址

颜色表中每个寄存器占3个字节，对应于红绿蓝三色的亮度值。每个寄存器的3个字节连续依次存放。
返回值：无

(10)子功能13h：选择颜色子集
输入参数：AH＝10h
AL＝13h
如果BL＝0：选择模式
这时　BH＝0：4个64色的子集
BH＝1：16个16色的子集
如果BL＝1：选择子集
这时　BH＝子集号(0～4或0～15)
返回值：无

一次完整的操作需调用两次该功能，首先选择模式然后选择子集。这一功能实际是对属性控制器的颜色选择寄存器进行操作。

(11)子功能15h：读单个DAC寄存器
输入参数：AH＝10h
AL＝15h
BX＝DAC寄存器号(0～255)
返回值：DH＝红色亮度值(0～3Fh)
CH＝绿色亮度值(0～3Fh)
CL＝蓝色亮度值(0～3Fh)

(12)子功能17h：读DAC寄存器组

输入参数：AH＝10h
AL＝17h
BX＝起始DAC寄存器号(0～255)
CX＝要读的寄存器数目(1～256)
ES:DX＝存放返回值的缓冲区地址
返回值：缓冲区中的数据

(13)子功能18h：设置PEL屏蔽

输入参数：AH＝10h
AL＝18h
BL＝PEL屏蔽值
返回值：无

该功能对应于属性控制器中的彩色位面允许寄存器。

(14)子功能19h：读PEL屏蔽

输入参数：AH＝10h
AL＝19h
返回值：BL＝PEL屏蔽值

该功能对应于属性控制器中的彩色位面允许寄存器。

(15)子功能1Ah：读子集状态

输入参数：AH＝10h
AL＝1Ah
返回值：BH＝当前颜色子集号
BL＝0：当前处于4子集模式
BL＝1：当前处于16子集模式

该功能与子功能13h相对应。

(16)子功能1Bh：将DAC寄存器转换成灰度

输入参数：AH＝10h
AL＝1Bh
BX＝起始DAC寄存器号(0～255)
CX＝要转换的寄存器数目(1～256)
返回值：无

该功能将一组寄存器从彩色值转换位灰度值，转换公式如下：

灰度＝30%红＋59%绿＋11%蓝

功能11h：装入字符发生器

其包含17个控制文本显示的子功能。

功能12h：取VGA状态

其包含一组彼此无关的子功能。

(1)子功能10h：返回VGA配置信息

输入参数：AH＝12h
BL＝10h

返回值：BH＝0　彩色方式有效
　　BH＝1　单色方式有效
　　BL＝存储器容量：0＝64K，1＝128K，2＝192K，3＝256K
　　CH＝特征位
　　CL＝EGA开关设置

(2)子功能20h：屏幕打印

输入参数：AH＝12h
　　BL＝20h

返回值：无

该功能将屏幕内容输出到打印机上，它只支持字符模式。

(3)子功能30h：为下一个文本模式选择扫描线总数

(4)子功能31h：允许/禁止在设置模式期间装入调色板

输入参数：AH＝12h
　　AL＝0　允许(缺省状态)
　　1　禁止
　　BL＝31h

返回值：AL＝12h　表示支持该功能

(5)子功能32h：允许/禁止VGA存取

输入参数：AH＝12h
　　AL＝0　允许存取VGA存储器及寄存器
　　1　禁止存取VGA存储器及寄存器
　　BL＝32h

返回值：AL＝12h　表示执行了该功能

(6)子功能33h：允许/禁止灰度求和

输入参数：AH＝12h
　　AL＝0　允许灰度求和
　　1　禁止灰度求和
　　BL＝33h

返回值：AL＝12h　表示支持该功能

(7)子功能34h：允许/禁止CGA/MDA光标仿真

(8)子功能35h：在显示器之间作转换

输入参数：AH＝12h
　　AL＝选择视频系统
　　0　原适配器上的视频系统关闭
　　1　原主板上的视频系统关闭
　　2　转换到未使用的视频系统
　　3　用ES:DX中的参数初始化视频系统
　　BL＝35h
　　ES:DX＝128字节的缓冲区地址

返回值：AL＝12h　表示支持该功能

(9)子功能36h：显示器开/关

输入参数：AH＝12h

AL＝0　允许屏幕刷新

1　禁止屏幕刷新

BL＝36h

返回值：AL＝12h　表示支持该功能

禁止屏幕刷新后，显示卡将停止对显示存储器的访问，这可使CPU对显示存储器的访问达到最快。

功能13h：写文本字符串

功能1Ah：读或写配置

这个功能分为两个子功能，分别读和修改系统中显示设备的当前配置。

(1)子功能0：读显示配置

输入参数：AH＝1Ah

AL＝0

返回值：AL＝1Ah

BL＝主显示配置代码

BH＝辅助显示配置代码

显示配置代码含义如下：

0　无显示

1　MDA

2　CGA

4　接彩色显示器的EGA

5　接单色显示器的EGA

6　PGC(专用图形控制器)

7　接单色显示器的VGA

8　接彩色显示器的VGA

0Bh　接单色显示器的MCGA

0Ch　接彩色显示器的MCGA

(2)写显示配置

输入参数：AH＝1Ah

AL＝1

BL＝主显示配置代码

BH＝辅助显示配置代码

返回值：AL＝1Ah

功能1Bh：返回VGA状态信息

输入参数：AH＝1Bh

BX＝0

ES:DI＝64字节缓冲区地址

返回值：AL＝1Bh
　　缓冲区中的数据

功能1Ch：保存/恢复显示适配器状态

这个功能分为三个子功能。

(1)子功能0：返回所需缓冲区的大小

输入参数：AH＝1Ch
　　AL＝0
　　CX＝要保存的数据类型
　　位0　寄存器
　　位1　BIOS数据区
　　位2　DAC寄存器

返回值：AL＝1Ch
　　BX＝所需缓冲区的大小(以64字节为单位)

(2)子功能1：保存显示适配器状态

输入参数：AH＝1Ch
　　AL＝1
　　CX＝要保存的数据类型
　　ES:BX＝缓冲区地址

返回值：AL＝1Ch
　　缓冲区中的数据

(3)子功能2：恢复显示适配器状态

输入参数：AH＝1Ch
　　AL＝2
　　CX＝要恢复的数据类型
　　ES:BX＝缓冲区地址

返回值：AL＝1Ch

2.3.3　VESA扩展BIOS

与标准VGA的BIOS功能一样，所有VESA扩展BIOS功能都是通过中断10H来调用，它们使用同一个功能号4Fh，该功能号在标准VGA BIOS中未用。

所有的VESA扩展BIOS功能都具有相同的调用格式，如下：

输入参数：AH＝4Fh
　　AL＝VESA功能号
　　……

返回值：AL＝4Fh　如果支持此功能
　　AH＝0　如果功能调用成功
　　1　如果功能调用失败
　　2～0FFh　保留(应视为失败)
　　……

下面分别介绍VESA BIOS的各个功能。

功能0：返回Super VGA信息

输入参数：AH＝4Fh

AL＝0

ES:DI＝256字节缓冲区地址

返回值：AL、AH

缓冲区中的数据

返回数据块的格式及含义如下：

4字节　指向VESA标志字符串的指针

1字节　VESA副版本号

1字节　VESA主版本号

4字节　指向OEM字符串的指针

4字节　板的功能

4字节　指向所支持模式数据块的指针

所支持模式数据块中依次存放当前显示卡所支持的VGA扩展模式的模式号，每个模式号占2个字节，该数据块以0FFFFh结尾。

功能1：返回Super VGA模式信息

输入参数：AH＝4Fh

AL＝1

CX＝所要查询的模式

ES:DI＝存放返回信息的缓冲区地址

返回值：AL、AH

缓冲区中的数据

返回数据的格式及含义如下：

2字节　模式属性

1字节　窗口A属性

1字节　窗口B属性

2字节　窗口粒度

2字节　窗口尺寸

2字节　窗口A段地址

2字节　窗口B段地址

4字节　换页操作功能地址

2字节　每条扫描线所占字节数

以下位可选信息：

2字节　水平分辨率

2字节　垂直分辨率

1字节　字符宽度

1字节　字符高度

1字节　存储器位面数

1字节　　每个象素所占位数
1字节　　组数
1字节　　存储器结构类型
1字节　　组尺寸

下面对有关信息作具体解释：

·模式属性　其16位数据各位的定义如下：

位0　现行显示器是否支持本模式，1表示支持
位1　本块可选信息是否有效，1表示有效
位2　是否支持BIOS文本功能，1表示支持
位3　彩色/单色模式，1表示为彩色模式
位4　图形/文本模式，1表示为图形模式
位5～位15　保留

·窗口A属性、窗口B属性　窗口指显示存储器所占用的系统内存地址，即视频地址空间。VGA可具有两段视频地址，这时即对应于A、B两个窗口。窗口属性字节各位的定义如下：

位0　窗口是否有效，1表示有效
位1　窗口是否可读，1表示可读
位2　窗口是否可写，1表示可写
位3～位7　保留

·窗口粒度　显示存储器分页的最小增量，其必须能被窗口尺寸整除，大多数情况下等于窗口尺寸。单位为1024字节。

·窗口尺寸　显示存储器页的大小，单位为1024字节。

·窗口A段地址、窗口B段地址　窗口在系统内存中的段地址。

·换页操作功能地址　可直接调用此地址出的代码来进行存储器换页，也可通过VESA BIOS功能5来进行换页，这种方法速度稍快，但兼容性要差一些。

·每条扫描线所占字节数　指在一个位面中所占的字节数。

·水平分辨率　每一显示行的象素数(图形模式)或字符数(文本模式)。

·垂直分辨率　每一显示列的象素数(图形模式)或字符数(文本模式)。

·字符宽度、字符高度　以象素为单位表示的一个字符的宽度和高度。

·存储器结构类型　指显示存储器的组织方式，有效的类型有：

0　文本模式
1　CGA图形模式
2　Hercules图形模式
3　4位面图形模式
4　压缩象素图形模式
5　非链结构4，256色图形模式
6～0Fh　保留
10h～0FFh　可有制造商定义

·组数、组尺寸　仅适于具有非线性映射关系的图形模式，如CGA图形模式。

功能2：设置VGA扩展显示模式

输入参数：AH＝4Fh

AL＝2

BX＝显示模式号

VESA模式的一般规则如下：

位0～位7　模式号

位8　VESA模式标志，0表示非VESA定义模式，1表示VESA定义模式

位9～位14　为将来扩充保留(以0填充)

位15　0：清显示存储器，1：保持显示存储器

返回值：AL、AH

VESA标准规定了各扩展模式的统一模式号，只有用这些统一的模式号及该功能调用才能实现标准的模式设置操作。

功能3：返回当前显示模式

输入参数：AH＝4Fh

AL＝3

返回值：AL、AH

BX＝当前显示模式号

功能4：保存/恢复VGA视频状态

该功能有三个子功能组成。

(1)子功能1：返回所需缓冲区大小

输入参数：AH＝4Fh

AL＝4

DL＝0

CX＝要保存的状态

位0　硬件状态

位1　BIOS数据

位2　DAC状态

位3　Super VGA状态

返回值：AL、AH

BX＝所需缓冲区的大小，以64字节为单位

(2)子功能1：保存VGA视频状态

输入参数：AH＝4Fh

AL＝4

DL＝1

CX＝要保存的状态

ES:BX＝缓冲区地址

返回值：AL、AH

(3)子功能2：恢复VGA视频状态

输入参数：AH＝4Fh

AL＝4
DL＝2
CX＝所保存的状态
ES:BX＝缓冲区地址

返回值：AL、AH

功能5：显示存储器换页

其依BH值的不同而分两种调用状态

(1)选择显示存储器页

输入参数：AH＝4Fh
AL＝5
BH＝0
BL＝窗口号(0：窗口A；1：窗口B)
DX＝页在显示存储器中的起始位置(以窗口粒度为单位)

DX应等于窗口尺寸除以窗口粒度再乘以页号，但通常窗口粒度就等于窗口尺寸，这时DX就直接等于页号。

返回值：AL、AH

为提高运行速度，可以通过对功能1所返回的地址执行远调用，来直接调用该功能，这时参数仍通过BL、DX传送，但不需要AH、AL、BH的值，且不返回任何信息。分页功能的地址依赖于显示卡、显示模式，甚至可能不存在，因此这种直接调用的方法不是很保险。

(2)返回当前显示存储器页

输入参数：AH＝4Fh
AL＝5
BH＝1
BL＝窗口号

返回值：AL、AH
DX＝当前页在新式存储器中的起始位置(以窗口粒度为单位)

这是VESA所提供的一个最有意义的功能，因为不同型号的VGA采用了不同的分页方法，而换页是访问扩展模式下显示存储器所必须进行的一项操作，该功能提供了通用的换页操作方法。

2.4　兼　容　性

各厂商所生产的VGA产品都在标准VGA的基础上进行了扩充，而各种产品所作的扩充都存在差别，这就使得它们在硬件上互不兼容，对程序员来说，各种VGA之间的差别主要存在于三个方面：模式号、分页方式、换页操作。这里介绍在编程中如何应付这些差别，采用VESA标准是最主要的方法。同时还将介绍采用Trident公司VGA芯片的显示卡在这三方面的情况，我国绝大部分兼容机上的显示卡采用的都是Trident公司的芯片，这些显示卡包括TVGA8900系列、TVGA9000系列及T9400系列等，将这些显示卡统称为Trident VGA。

2.4.1 模式号

各VGA厂商在他们的产品中增加一些新的显示模式时，他们必须给每一个新模式规定一个模式号，以使应用程序能明确地设置出每一种模式，这只是一个很小的问题，但各VGA厂商在这一问题上仍然没有达成统一，同一显示模式在不同的VGA上具有不同的模式号，Super VGA的混乱程度由此可见一斑。最终由VESA标准给各种扩展模式规定了统一的模式号， 现在各种VGA在支持VESA标准的同时也保留了自己的一套模式号，在VGA上可以用两个不同的模式号来设置同一个扩展显示模式，但只有采用VESA模式号及VESA的BIOS模式设置功能，才能实现标准的模式设置操作，才能保证程序能在各种不同的VGA上正确运行。表2-3列出了各种VGA图形模式的VESA模式号及其在Trident VGA上的模式号。

表2-3 VGA模式号

VGA图形模式	VESA模式号	Trident VGA模式号
640× 480× 16色		12h
800× 600× 16色	102h	5Bh
1024× 768× 16色	104h	5Fh
320× 200×256色		13h
640× 400×256色	100h	5Ch
640× 480×256色	101h	5Dh
800× 600×256色	103h	5Eh
1024× 768×256色	105h	62h
320× 200×32K色	10Dh	7Eh
512× 480×32K色	170h	70h
640× 480×32K色	110h	74h
800× 600×32K色	113h	76h
320× 200×64K色	10Eh	7Fh
512× 480×64K色	171h	71h
640× 480×64K色	111h	75h
800× 600×64K色	114h	77h
320× 200×16M色	10fh	6Bh
640× 480×16M色	112h	6Ch

640×480×16色和320×200×256色是标准VGA的图形模式，在各种VGA上它们都具有相同的模式号，分别为12h和13h，因此VESA没有再给这两个模式规定模式号。VESA模式号都大于100h，而各种VGA自有的模式号都小于100h。当使用VESA模式号时，必须采用VESA BIOS功能2进行模式设置；当使用VGA自有的模式号时，只能采用标准VGA BIOS功能0进行模式设置，在某些VGA上则必须使用其特有的BIOS功能进行模式设置。

2.4.2　分页方式

由于位面技术的采用，在标准VGA显示模式下显示存储器不需要分页，显示存储器分页产生于Super VGA，而各种VGA采用了不同的分页方式。被采用的分页方式有以下几种：

1. 64K单页

每页的大小为64K，显示存储器占用的主机内存地址空间为：A0000H～AFFFFH。这种分页方式与标准VGA完全兼容，它是最常用的一种分页方式。

2. 128K单页

每页的大小为128K，视频地址空间为：A0000H～BFFFFH。PC机留给视频系统的地址空间正好为这128K，但通常的VGA都考虑了同时对两个显示设备的支持，一个用于图形显示，一个用于字符显示，因而一般的VGA在图形模式下只使用了从A0000H开始的64K的地址空间，而将另一部分地址空间用在字符模式下，而这种分页方式使用了全部地址空间，因此其它显示设备不能与之共同使用。一些专业的图形系统往往需要接双显示器，因此这种分页方式采用得较少。Trident在其最早的8800BR芯片中采用了这种分页方式，但在其以后的芯片中都采用了第1种分页方式。

在页粒度为64K时，这种分页方式可以完全兼容第1种分页方式。

3. 64K双页，64K地址空间

存在两个64K的存储器页，一个只写，一个只读，两个页占用同一块主机内存地址空间：A0000H～AFFFFH，但它们可以映射到显示存储器的不同位置，当向视频地址进行写操作时，数据将写到只写页所对应的显示存储器位置，当进行读操作时，数据将来自于只读页所对应的显示存储器位置。双页方式能使图象拷贝操作的速度加快一倍，这是其得到使用的主要原因，在很多VGA中都采用了这种双页方式。

将两个页始终映射到同一块显示存储器上，即可实现与第1种分页方式的兼容。

4. 64K双页，128K地址空间

有两个64K的存储器页，它们都可读可写，它们使用不同的视频地址空间，分别为：A0000H～AFFFFH和B0000H～BFFFFH。与上一种双页方式相比，这种可读可写的双页方式的好处在于，在进行图象拷贝时能对图象数据进行逻辑运算。同样，它与字符模式的地址空间相冲突，无法接双显示设备，因而用得较少。

始终只使用第一个页进行操作，就可与第1种分页方式保持兼容。

5. 32K双页

有两个32K的存储器页，都可读可写，它们使用的视频地址空间分别为：A0000H～A7FFFH和A8000H～AFFFFH。这种分页方式很少使用。将两个页依次映射到连续的64K显示存储器上，即可与第1种方式兼容。

不同的分页方式所带来的编程上的差异是很大的，使用VESA BIOS功能能够检测出各种分页方式，但它不能消除各种分页方式在编程上的差异。针对每一种分页方式编写多个那些需要直接访问显示存储器的子程序，然后再根据检测结果来调用某个子程序，这种程序实现方式无疑是程序员所不希望的。所幸的是，其它各种分页方式都能兼容第1种分页方式，而且采用其它分页方式的VGA大都以默认状态提供了对第1种方式的兼容，也就是说，如果一个程序不是特意要使用其它分页方式，它就只会处于第1种分页方式之下。这就使得我们只针

对第1种分页方式编程，就能得到一个通用的程序，虽然这将放弃对可能存在的双页功能的利用，但能大大减小程序的复杂程度，而且在大多数应用中双页方式的优点都是不必要的。

对大多数软件来说，应付分页方式差别的最好方法就是，只按64K单页方式编程。

2.4.3 换页操作

由于位面技术的采用，标准VGA上的各种显示模式都不需要换页，换页操作产生于Super VGA，所以各种VGA都具有不同的换页操作方法，换页一般通过寄存器操作或BIOS功能调用来实现，VESA标准提供了统一的BIOS换页功能，但在直接操作寄存器时仍然没有标准，而后者具有更快的速度。下面分别介绍标准的换页操作方法及Trident VGA自己的换页操作方法。

1. 标准的换页操作

采用VESA的BIOS换页功能是实现标准换页操作的唯一方法。VESA BIOS提供了两种换页操作方法，一是，使用VESA BIOS功能5，这种方法最为标准；二是，首先用VESA BIOS功能1获得换页功能的地址，然后直接调用该地址下的换页功能，这种方法能获得较快的速度。这两种方法的具体实现方式见2.3.3。

2. Trident VGA的换页操作

Trident VGA提供了两种换页操作方法，下面分别介绍。

(1)寄存器操作

Trident VGA上新增了一个方式控制寄存器，用来进行换页操作，其使用的I/O地址及操作方法与标准VGA的定序器寄存器组相同，即索引端口地址为3C4h，数据端口地址为3C5h，其索引值为0Eh。在将页号写入寄存器之前需将页号的位1取反，然后写入，而读出的则是正常值，不需取反。

(2)BIOS功能调用

Trident VGA提供了一组自己的BIOS功能，其使用的功能号为70h，其子功能4可用来获得换页子程序的地址，该功能的调用格式如下：

输入参数：AX＝7000h

BX＝4

返回值：ES:DI＝换页子程序的地址

首先将页号放于DL寄存器，然后对所获得的地址执行远调用，即可实现换页操作。

2.4.4 显示存储器容量检测

VGA所配置的显示存储器容量往往是不固定的，如TVGA8900系列显示卡，其标准配置为512K，但可扩充到1M。9400系列的标准配置为1M，可扩充到2M。显示存储器容量直接决定了其所能支持的显示模式，在那些支持多显示模式的软件中往往就需要对显示卡的存储器容量进行检测。标准VGA的BIOS提供了检测显示存储器容量的功能，但其最大只能检测256K，在现在这已没有什么实际意义。下面介绍利用VESA BIOS进行检测的方法及Trident VGA的检测方法。

1. 用VESA BIOS检测容量

VESA BIOS没有直接提供检测显示存储器容量的方法，但利用其功能0和功能1可间接检测出显示卡所配置的存储器容量。

首先用功能0获得的显示卡所能支持的所有显示模式，通常VESA只返回那些为显示存储器容量所支持的模式；用功能1逐一获得每种模式的详细信息，将其中所提供的：每条扫描线所占字节数、垂直分辨率、存储器位面数，三个数据相乘，即可得到每种显示模式所需的存储器容量，但垂直分辨率、存储器位面数是可选信息，有可能得不到它们；或者预先计算出每种可能出现的扩展模式所需的存储器容量，将它们放在程序中，由此直接得到每种模式所需的存储器容量；从各种模式所需的存储器容量中找出最大值，将该值向上取整到256K乘2的n次方，即得到当前显示卡的存储器容量。

这可能是唯一通用的检测方法，但它不是被直接提供的，所以在某些卡上可能得不到正确的结果。

2. Trident VGA的容量检测方法

Trident VGA提供了两种检测方法，下面分别介绍。

(1)寄存器操作

Trident VGA上有一高速缓存寄存器，可用来检测显示存储器容量，其使用的I/O地址及操作方法与标准VGA的CRT控制器寄存器组相同，即索引端口地址为3D4h，数据端口地址为3D5h，其索引值为1Fh。该寄存器的低两位表示了显示存储器的容量，从0～3依次对应于256K，512K，1M，2M。

(2)BIOS功能调用

Trident VGA的BIOS功能0也可用来检测显示存储器容量，该功能的调用格式如下：

输入参数：AX＝7000h

　　BX＝0

返回值：AL＝70h(如果支持)

　　CL＝显示器类型

　　CH＝状态

　　位0　6845仿真允许

　　位4　VGA保护允许

　　位6～位7　显示存储器容量

　　00　256K

　　01　512K

　　10　1M

　　11　2M

　　DX＝适配器标识符

　　DI＝BIOS版本

2.4.5　其它兼容性问题

除上述几方面之外，VGA上还存在其它一些兼容性问题：某些VGA提供了一些特殊的显示模式，如2色模式、4色模式及一些特殊的分辨率，如720×540、768×1024等，这些模式不被VESA标准所支持；一些早期的VGA在256色模式中采用了特殊的存储器组织方式，主要有两种，一是采用位面技术，每个象素在每个位面中占2位；二是采用非线性的地址映射关系，

相邻的四个象素分别放在4个存储器页或4个位面中。这些问题的影响要小得多，在编程中可以完全忽略这些问题。

在VESA标准未得到广泛支持时，主要采用这样一种方式来编写使用VGA扩展模式的通用软件，即：将软件中那些直接与VGA打交道的功能集中起来，放在一个独立的图形驱动文件中，然后给每种显示卡编写一个图形驱动文件，或者公布图形驱动文件的格式，让VGA厂商来提供这些图形驱动文件，从而使软件能在各种VGA上运行。显然只有那些极具实力的软件公司才能采用这一方法，而大多数软件开发人员就只能放弃对VGA扩展模式的使用。现在，VESA标准所获得的广泛支持，终于使众多的程序员都具有了使用VGA扩展特性的能力，本书正是基于VESA标准来介绍VGA的通用编程技术的。

第 3 章　程序设计基础

3.1　程序设计语言

从这一章开始即要涉及到具体的编程，这里对本书所采用的程序设计语言作一个概要的介绍，以便于阅读本书的程序。

3.1.1　C++

本书所要实现的并不是一些单独的示例性程序，而是一套完整的程序系统，这就需要唯一选定一种程序设计语言，本书采用了C++，并且全面使用了C++的面向对象机制。

在本书的编程中主要用到了如下一些C++特有的语法成分：引用；new、delete操作符；类(class)；静态成员；类的构造函数及析构函数；继承、多重继承；虚函数；友元；函数重载；操作符重载。如果需要阅读和修改本书的程序，就需要掌握C++的这些语法成分。

程序中用到了如下一些C++的库函数：

int86()	调用8086软中断
intr()	调用软中断
fopen()	打开一个文件
fwrite()	向文件中写数据
fread()	从文件中读数据
fseeK()	移动文件指针
fclose()	关闭一个文件
findfirst()	在磁盘中查找一个文件
remove()	删除一个文件
malloc()	分配一块内存
calloc()	分配一块内存，并清零
free()	释放一块内存
memcpy()	拷贝内存中的数据
memmove()	拷贝内存块中的数据(可重叠)
coreleft()	返回自由内存的大小
getvect()	读取中断向量
setvect()	安装一个中断功能
disable()	屏蔽中断
enable()	开中断
inportb()	从端口读入一个字节
outportb()	输出一个字节到端口
delay()	延迟若干毫秒
biosprint()	调用BIOS打印接口(INT 17H)
biosKey()	调用BIOS的键盘接口(INT 16H)

biostime()	读取或设置系统时钟(INT 1AH)
strcpy()	拷贝字符串
strlen()	获取字符串的长度
vsprintf()	格式化输出到串中
MK_FP()	根据段地址和偏移地址生成一个远地址
sqrt()	求平方根
abs()	求绝对值
itoa()	将整数转换为字符串

3.1.2 嵌入汇编

在CGA、EGA及早期的VGA上，那些直接对显示卡进行操作的程序都要求必须用汇编语言编写，现在虽然已不再有这种要求，但速度仍然是图形显示中必须考虑的一个重要因素，那些直接对VGA硬件进行操作的程序最好还是直接用汇编语言来实现。但采用纯粹的汇编语言来实现本书的某些程序，就要涉及到混合语言程序设计问题，这会给整套程序的组织和维护带来很大的麻烦。所幸的是，C++和C都提供了一种嵌入汇编机制，它允许直接在函数中加入汇编代码，本书即采用了嵌入汇编来实现那些直接进行硬件操作的程序。

使用嵌入汇编既保证了程序的效率，又不会给整套程序的结构带来影响，此外对那些不熟悉汇编语言的读者来说，掌握嵌入汇编比掌握一套独立的汇编语言要容易得多，这就使得他们能够较容易地具备对本书的程序进行维护和修改的能力。

当需要对本书的程序进行修改和扩充时，对嵌入汇编的掌握是必须的。嵌入汇编并不是一种独立的语言，它实际上是提供了一种在C++或C中直接使用汇编指令的方式。掌握嵌入汇编，主要是要了解8086的指令集和寄存器，但并不需要对8086的指令有全部的了解，只需要了解那些要用到的指令。各种版本的C++或C程序员手册都对嵌入汇编有专门的介绍，8086的指令集和寄存器则可在各种介绍汇编语言的书中查到。

下面对程序中所用到的汇编指令作简单的介绍。

(1) MOV OD1,OD2

将操作数OD2中的一个字节或一个字传送到操作数OD1中。在C++的嵌入汇编中，操作数除了可以是汇编语言中所规定的类型外，还可以是C++程序中所定义的各种变量，但不能是类的数据成员。

(2) PUSH OD

将操作数OD中的一个字放入堆栈的栈顶。

(3) POP OD

将堆栈栈顶的一个字传入OD中。

(4) IN OD1,OD2

从地址为OD2的端口读入一个字或一个字节放入OD1。

(5) OUT OD1,OD2

将OD2中的一个字或一个字节输出到端口地址OD1。

(6) LDS OD1,OD2

获取OD2的远地址，段址放在DS寄存器中，偏移量放在OD1中。

(7) LES OD1,OD2

获取OD2的远地址，段址放在ES寄存器中，偏移量放在OD2中。

(8) ADD OD1,OD2

OD1与OD2相加，结果放在OD1中。

(9) INC OD

OD加1。

(10) SUB OD1,OD2

OD1减去OD2，结果放在OD1中。

(11) DEC OD

OD减1。

(12) MUL OD

OD中的一个字节或一个字与AL或AX中的数相乘，双倍长度的结果放到AX或DX与AX中。

(13) NOT OD

OD按位取反。

(14) AND OD1,OD2

OD1、OD2相与，结果放到OD1中。

(15) OR OD1,OD2

OD1、OD2相或，结果放到OD1中。

(16) SHL OD,m

OD左移m位，m或为1，或为CL寄存器。

(17) SHR

OD右移m位，m或为1，或为CL寄存器。

(18) REP

重复前缀。使一条串操作指令重复执行若干次，重复次数放在CX寄存器中，每重复一次DI和SI寄存器中的数减1(方向标志DF=1)或加1(DF=0)。CLD指令可使DF=0，STD指令使DF=1。

(19) MOVSB/MOVSW

将内存地址DS:SI处的一个字节(MOVSB)或一个字(MOVSW)传送到内存地址ES:DI处，该指令可带重复前缀。

(20) STOSB/STOSW

将AL中的一个字节(STOSB)或AX中的一个字(STOSW)传送到内存地址ES:DI处，并可带重复前缀。

(21) INT n

调用中断向量号为n的软中断功能。

(22) CALL OD

调用一个子程序，OD为所要调用程序的地址。

(23) CMP OD1,OD2

OD1减OD2，结果不返回。该指令的操作结果对一些标志位产生影响，它主要与一些条件转移指令结合使用。

(24) JMP OD

无条件转移到OD处，OD为一标号，在嵌入汇编中，标号必须是C语句的标号，而不能是汇编指令段中的标号。

(25) JE/JZ OD

上一条指令(往往是CMP)中两个操作数相等或操作结果等于0，则转移到OD处。

(26) JNE/JNZ OD

上一条指令中两个操作数不等或操作结果不为0，则转移到OD处。

(27) JC OD

上一条指令中两个操作数相加或相减而发生进位或借位时，转移到OD处。

(28) JGE OD

上一条指令中OD1大于等于OD2，或操作结果大于等于0时，转移到OD处。

3.1.3 程序编写说明

本书提供的源程序分为三类，一是系统程序，这些程序构成了本书所要实现的VGA基础图形支持系统；二是应用程序，这是一些与本书的内容相关且有一定实用价值的程序；三是示例程序，这些程序用来对系统程序及某些技术的使用方法进行说明。这三类程序在书中将分别进行编号。

系统程序和应用程序都来自于本书所附带的一张软盘，但软盘中的程序很少有注释，而书中则增加了很多注释。书中没有全部列出该软盘中的源程序，只是列出了主要部分，因此书中给出的系统程序不能组成完整的程序系统。这些来自软盘的程序，在书中都将给出它们在软盘中的文件名。软盘中的很多文件在书中都被分成几个部分在多处列出。有关本书所附带软盘的详细说明见第15章。

当对一段源程序进行注释时，注释处于该段程序之上，并采用C注释符“/* */”；当对一行源程序进行注释时，注释处于该行之后或之下，这时采用C++注释符“//”；在整段源程序之后将给出那些较复杂的说明。当对一段程序进行注释时，往往会指明注释的行数，该行数为书写行，它包括了那些无实际语句和指令的行，如空行、大括号等。

3.2 程序系统的内容及构成

本书将要实现一套完整的VGA基础图形支持程序，书中所介绍的所有技术都将在这套程序中实现。与书中所介绍的各部分技术相对应，这套程序分为六个部分，这六部分程序的基本功能及相互关系如下：

·图形显示程序 这部分程序将实现对18种VGA图形模式的支持，提供每种图形模式下的各种基本图形显示功能，包括：写点，读点，画扫描线，读、写扫描线，清屏，画直线，画圆、椭圆、扇形，各种区域填充，读块、写块等。 这是整套程序中最基本也是最主要的一部分程序，其它各部分程序都将用到这部分程序。

·字符显示程序 这部分程序将提供对5类21种字库的字符显示功能的支持，所支持的字库包括：16点阵、24点阵英文字库；Borland C++十种字体的英文字库；一种轮廓矢量英文字库；16点阵汉字库；宋、黑、楷、仿宋四种字体的24点阵、32点阵、40点阵汉字库；UCDOS 3.0/3.1 26种字体的矢量汉字库。同时还将对小汉字库的构造和操作提供支持。字符

显示程序使用图形显示程序的有关功能来在各种图形模式下进行字符显示，并能使用下面所要介绍的图象缓存操作程序来实现字符打印。

·图形打印程序　这部分程序分为图象缓存操作程序和图象打印程序，应用程序使用图象缓存操作程序来完成其图形输出，然后用图象打印程序打印出图象缓存区中的图象。图象缓存操作程序继承于图形显示程序，因而它具有图形显示程序的所有功能，并具有字符输出功能。图象打印程序实现了在EPSON和HP两种系列打印机的8种分辨率下的图象打印功能。这部分程序还实现了各种图形模式下的屏幕硬拷贝功能。

·鼠标操作程序　系统中的鼠标驱动程序只能在两种标准的VGA图形模式显示鼠标光标，这部分程序实现了在VGA的各种扩展图形模式下显示和维持鼠标光标的功能，并实现了一组通用的鼠标操作接口程序，该接口程序能生成各种类型的鼠标事件，并为应用提供各种辅助的鼠标操作功能。这部分程序还提供了一个键盘模拟鼠标功能。这部分程序使用图形显示程序中的有关功能来进行鼠标光标的显示和维持。

·屏幕漫游程序　这部分程序实现了与屏幕漫游有关的虚屏定义、屏幕分割及屏幕移动功能，同时提供了鼠标自动漫游和键盘自动漫游功能，这部分程序继承于图形显示程序，因而能在漫游模式下使用所有的图形显示功能、字符显示功能、鼠标操作功能及屏幕硬拷贝功能。

·扩展内存操作程序　这部分程序提供了使用扩展内存管理规范(XMS)来操作扩展内存的功能。在图形显示程序、字符显示程序、图形打印程序中都要涉及到一些大容量的数据操作，在这些操作中程序实现了对扩展内存的自动支持。

由于C++所提供的良好的可重用性和可维护性，以及C++模块之间的弱耦合性，使得我们可以对上述这些程序分别进行设计和实现，而不需要从整体上对各部分程序之间的联系进行过多的考虑，但这种分别的设计应按一定的次序进行，即应先设计实现那些被使用或被继承的程序。下面将首先实现图形显示程序，这是本书中最大的一部分程序，将在第3章至第7章及第9章、第10章中得到完整的实现，第8章将实现扩展内存操作程序，其余4部分程序将在第11章至第14章分别实现。在每一章中，都将首先介绍相应的技术和原理，然后介绍程序的设计和实现。

上述所有程序最终将组合在一起，形成一套完整的VGA基础图形支持程序，以类库的形式提供给C++程序员使用。

3.3　图形显示程序设计

3.3.1　图形显示功能

这里概要介绍在程序中所需实现的每一项图形功能。这些功能可分为三类，一是基本图形操作功能，在实现这类功能时需要直接操作VGA的硬件，它们的实现方式与图形模式有关；二是基本绘图功能，这些功能的实现以基本图形操作功能为基础，它们不需要直接操作VGA，它们通过调用各种图形模式下的基本图形操作功能，来实现在各种模式下的图形显示。三是一些辅助功能。下面分别介绍这三类功能。

1. 基本图形操作功能

·写点 将某一坐标位置处的单一象素点的颜色值写到显示存储器中。这是最基本的一项图形显示功能，原则上说，其它的图形显示功能都可以该功能为基础来得到实现，但这样会使某些功能的速度显得太慢，所以还需要其它一些基本图形操作功能，以尽量提高图形显示的速度。这一功能主要供画直线、画圆、画椭圆、画扇形及带图案的区域填充等绘图功能使用，它也直接供应用程序使用。

·读点 从显示存储器中读出某一坐标位置处的一个象素点的颜色值。这也是一项最基本的功能，在很多情况下，它的速度显得太慢，因此需要具有其它一些读显示存储器的方式。它主要在任意区域填充中判别区域边界时使用，在图形打印程序中它还用于屏幕硬拷贝。

·画扫描线 将一条水平直线上所有象素点的颜色值写入到显示存储器中，扫描线上所有象素点具有相同的颜色值。扫描线上的象素点在显示存储器中是连续存放的，因此直接实现这项功能，比通过写点操作来实现这项功能具有更快的速度，在扫描线足够长时，这两者的速度会相差十倍。这一功能主要供填充功能使用，它也可直接供应用程序使用。

·读扫描线 从显示存储器中读出一条水平直线上所有象素点的颜色值，每个象素点具有不同的颜色值，读出后要分别保存它们。可以用读点操作来实现它，但直接实现它能获得快得多的速度。

·写扫描线 将读扫描线功能所读出的扫描线数据写到显示存储器中。可以用写点操作来实现它，但直接实现它能获得快得多的速度。这与读扫描线是一对相互关联的功能，它们主要在块保存、块恢复功能中使用。

·清屏 分为置零清屏和置色清屏两项功能，它们分别将当前显示模式所占用的显示存储器全部清零或置为某一颜色值。可以用画扫描线操作来实现这一功能，而直接实现它可以提高速度，但速度的提高不是很大。

2. 基本绘图功能

·画直线 在给定的两个端点间画出一条直线。

·画圆 根据给定的圆心和半径画出圆的边线。

·画椭圆 根据给定的圆心及长、短轴画出椭圆的边线。

·画扇形 根据给定的圆心、半径及角度范围画出扇形的边线

在光栅图形系统中，上述这些功能所生成的几何图形都由一些离散的象素点所构成。这些功能的主要任务就是计算出图形上所有象素点的坐标位置，然后调用写点功能来画出每个点。在确定图形中每个点的位置时，一般并不采用通常的计算方式，而需要采用一些能避免或较少使用乘除运算及浮点运算的专门的算法，以获得更快的速度。也可以直接实现这些功能，而不调用写点功能，以提高速度，但这会大大增加程序的复杂程度，而且使这些功能与图形模式有关，在需要支持多种图形模式时，这会使整个程序系统变得很庞大。

·画矩形 根据给定的左上端点及右下端点画出矩形的边框。该功能调用画直线功能来实现屏幕显示。

·画多边形 根据所给定的若干个端点，画出以这些端点为顶点的多边形的边框。该功能调用画直线功能来实现屏幕显示。

• **区域填充**　这是一大类功能，它包括画实圆、画实椭圆、画实扇形、画实矩形、画实多边形，任意区域填充等功能。这类功能还需支持带图案的填充。这类功能通过调用画扫描线功能和写点功能来实现屏幕显示。

• **块操作**　包括读块、写块两项功能，读块用于读出一个矩形区域中所有象素点的数据，写块用于将读块所读出的数据写到一个矩形区域中。这两个功能分别通过读扫描线、写扫描线功能来实现。读块、写块功能相结合即可实现块拷贝功能。有时一个块会具有很大的数据量，因此在该功能中要能够用扩展内存和硬盘来保存块数据。

3. 辅助功能

• **模式设置**　设置图形显示模式。这是进行任何图形显示之前首先必须完成的一项工作。

• **参数设置**　设置当前图形模式下与图形操作有关的一些参数。

• **关闭图形模式**　在程序退出前将显示模式设置为缺省的字符模式。

• **显示存储器换页**　这是大多数图形模式下，访问显示存储器所必须使用的一项功能。各个基本图形操作功能都将使用这一功能。

• **屏幕软关闭**　禁止屏幕刷新，使CPU对显示存储器的访问达到最快。在清屏时可使用这一功能。

• **调色**　包括写调色板、读调色板、写DAC色彩查找表、读DAC色彩查找表四项功能，其中前两项功能仅在16色模式中有用，后两项功能只在16色、256色模式中有用。在直接色彩模式下不需要使用这一功能。

3.3.2　功能与图形模式的关系

1. 影响编程的图形模式因素

要在多种图形模式下实现上述各项功能，图形模式对各项基本图形操作功能的实现方式有着直接的影响，这种影响主要来自三个方面：

(1)色彩模式

不同的色彩模式具有不同的显示存储器结构和不同的显示存储器访问方式，因此在不同的色彩模式下各种基本图形操作功能必须分别进行编程。这是影响图形操作编程的最主要因素。

(2)显示存储器分页

每一种色彩模式都具有多种分辨率，在不同的分辨率下，显示存储器分页会出现三种不同的情况：

• 不分页，屏幕上所有象素点的数据都处于显示存储器的同一页中。

• 行外分页，分页位置只会在两条扫描线之间，这时每条扫描线上的所有点都处于同一页中。

• 行内分页，分页处于扫描线上，这时某些扫描线上的点会处于相邻的两页。

表2-1列出了各种显示模式下存储器分页的情况。这三种情况中，按较后的情况编写的程序总是能适用于较前的情况，反之则不行，但较后情况的编程较复杂，速度也较慢。如果以速度为唯一目标，那就应区分这三种情况分别进行编程，但这会提高系统的复杂程度，增加编程和维护的难度，而事实上这样做所能获得的速度提高的幅度是非常微小的。本书的编

程将不区分这三种情况，即在每种色彩模式下都针对最不利的情况进行编程，这通常是第3种或第2种情况。这样，同一种色彩模式下的所有图形模式都使用同一组基本图形操作程序。

(3)图形模式的参数

指模式号、水平分辨率、垂直分辨率、每条扫描线所占字节数、所需的存储器总页数等参数。在每种图形模式中的这些参数都是不同的，将这些参数存放在一些变量中，即可消除它们对编程的影响。

2. 功能分类

这里根据各项功能在编程上与图形模式的相关程度对它们分类，这一分类是确定程序结构的基本依据，分为如下四类。

(1)与图形模式完全相关

指在每一图形模式下编程都存在差别的功能，在每一图形模式中都要编写一个实现这类功能的程序。这类功能只有一项：参数设置功能。在每种图形模式下，与图形操作有关的参数都具有不同的值，该项功能用来设置每种图形模式下的有关参数。

(2)与色彩模式相关

这些功能在不同的色彩模式下需分别进行编程，而属同一色彩模式的各图形模式共用一组程序。这类功能包括所有基本图形操作功能，其中置零清屏功能只与色彩模式部分相关，即某些色彩模式的置零清屏功能是相同的。

(3)与图形模式完全无关

在所有的图形模式下这些功能的编程是完全相同的，在整个程序系统中这些功能只应实现一次。这类功能包括：关闭图形模式、显示存储器换页、屏幕软关闭、调色。其中调色功能在直接色彩模式下是无用的。

(4)与图形模式无直接关系

所有的基本绘图功能都属于这一类。这类功能本身是与图形模式无关的，但在不同的色彩模式下它们需要调用不同的基本图形操作功能，因此它们与色彩模式间接相关。在程序系统中，这些功能最好只被实现一次，因为实现这些功能的程序量较大，如果在系统中存在这些功能的多个版本，那将给程序维护带来很大的麻烦。避免在各色彩模式下多次编写基本绘图程序，是程序设计中所要解决的一个最主要问题，采用C++的继承及虚函数机制能够很有效、很方便地解决这一问题。

3.3.3 颜色处理

1. 颜色表示方式

在不同的色彩模式下颜色变量的数据格式是各不相同的，16色、256色模式下的颜色变量为1个字节，高彩色模式下为2个字节，真彩色模式下为3个字节，很多功能都需要将颜色值作为输入或输出参数，因此有必要采用一种统一的数据格式，来表示各种色彩模式下的颜色值。

在程序中定义了一个联合(union)变量，在各种色彩模式下都统一采用该联合变量来表示颜色值，该联合的定义如下：

系统程序3-1　VGABASE.H

```
union COLOR {
     long dword;  //用于清零，并保留给可能出现的32位真彩色模式
     int word;  //用于存放高彩色模式下的颜色值
     //其格式与高彩色模式下显示存储器中的数据格式一致
     unsigned char byte;  //用于存放16色、256色模式下的颜色值
     unsigned char rgb[3];  //用于存放真彩色模式下的颜色值
     //三基色的存放顺序为蓝、绿、红，这与显示存储器中的格式相一致
     };
```

由于采用了该联合变量，程序中增加了一项颜色转换功能，用于将各种色彩模式下的原始颜色变量设置为该联合变量。这一功能与色彩模式相关，而且在15位色和16位色模式下也是不同的。

采用该联合变量的另一个好处是，在基本图形操作编程中统一了15位色和16位色的颜色值格式，使得这两种色彩模式能共用一组基本图形操作程序。

2. 颜色传递方式

所有的图形显示功能都需要将颜色值作为一个输入参数，将颜色值传递给图形显示功能的方式通常有两种，一是直接传递，即每次调用一个图形显示功能时都要给定一个颜色值；二是间接传递，即在系统中维持一个当前颜色值，图形显示功能总是用当前颜色值来显示图形，而不需要在调用时传递颜色值。当应用程序频繁改变图形显示的颜色时，就宜于采用直接传递方式，当应用程序用一种颜色连续多次进行图形显示时，则宜于采用间接传递方式，但很难估计应用程序会更偏向于哪种图形显示方式，所以也就很难确定这两种颜色传递方式的优劣。

程序中的所有图形显示功能都提供了对这两种颜色传递方式的支持，但直接实现的是间接传递方式，因为在使用COLOR联合来表示颜色的情况下，间接传递方式更便于应用程序使用，而采用直接传递方式的程序通过调用间接传递方式的程序来实现，因此它们的速度较慢。

由于采用了间接传递方式，程序中维持了一个用COLOR联合变量来存储的当前颜色值，因此程序中增加了一项设置当前颜色值功能，其用原始颜色变量来设置保存在COLOR联合变量中的当前颜色值，该功能与色彩模式相关，而且在15位色和16位色模式中存在差别。

3.3.4　编程方案

1. 概要

每项图形功能用一个函数来实现。

对应于每一种图形模式定义一个类，称为图形模式类，一种图形模式下所有功能的函数都包含在相应的图形模式类中。

属于同一种色彩模式的那些图形模式类，它们的很多函数都是相同的，没有必要在这些图形模式类中重复定义那些相同的函数，因此针对每种色彩模式也定义了一个类，称为色彩模式类，它们包含了那些同一色彩模式下的公用函数，每个图形模式类都派生于一定的色彩

模式类，通过继承，图形模式类获得了色彩模式类中的公用函数。对高彩色模式和间接色彩模式在两个层次上定义了相应的色彩模式类。

还存在一些函数，它们在所有的色彩模式中都是相同的，因此进一步定义了一个基本图形类，该类包含了那些为所有色彩模式所共用的函数，每个色彩模式类都派生于该基本图形类，通过继承，所有的色彩模式类及图形模式类都获得了基本图形类中所定义的那些公用函数。

按照上述归纳，可形成一个具有4级层次的类体系，称图形显示类体系。该类体系能够以最小的代码冗余实现一套支持各种图形模式的图形显示程序。

2. 图形显示类体系的层次结构

所实现的类体系的层次结构如图3-1所示。图中每个方框代表一个类，方框中的字符串即为程序中所使用的类名，其中带虚框的类为抽象类。

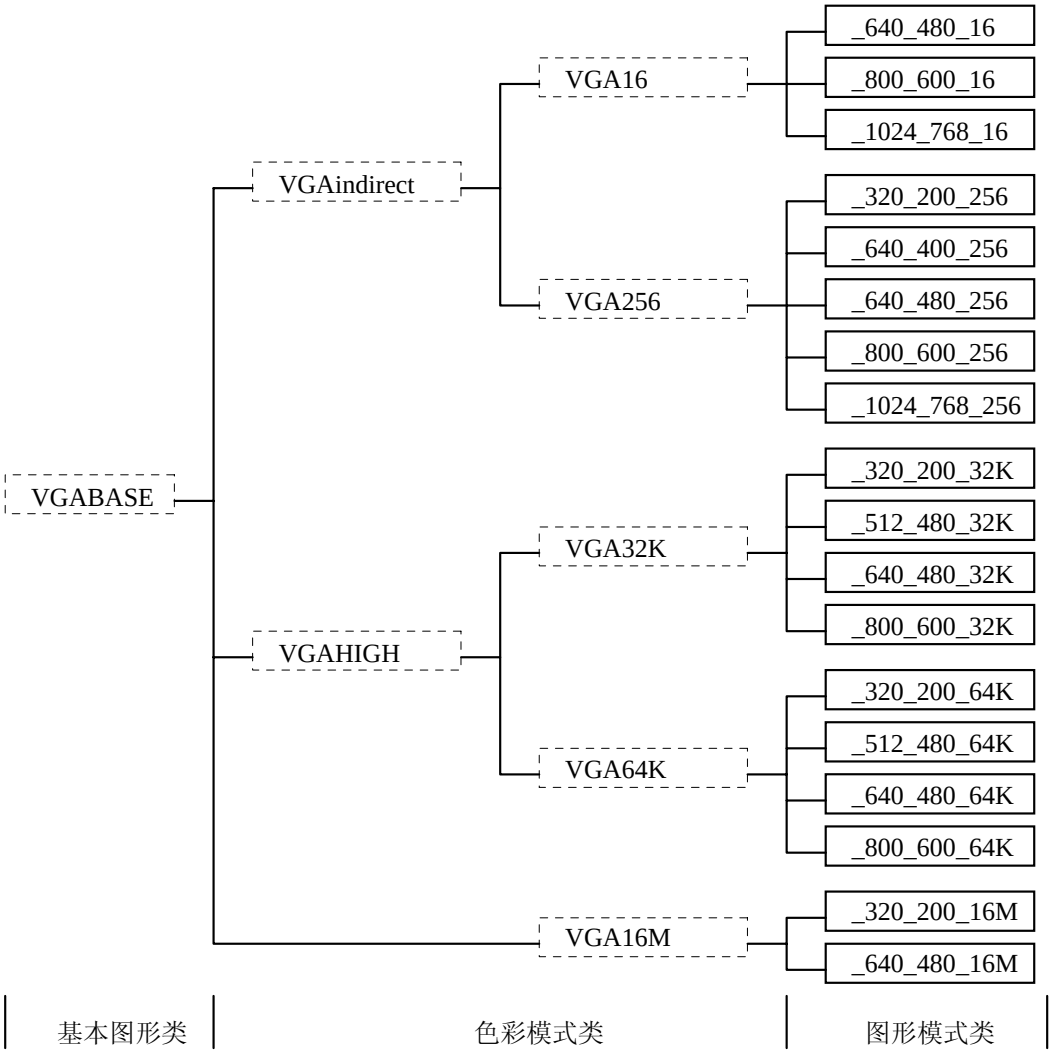

图3-1 图形显示类体系

下面对图形显示类体系中的各个类作简要说明

• class VGABASE　基本图形类。是其它所有类的基类。在该类中说明了整个类体系中的绝大多数函数及全部数据，该类主要实现了与图形模式完全无关和与图形模式无直接关系的函数，那些未在该类中实现的函数被说明为纯虚函数，留待该类的派生类来实现。该类按常用方式实现了置零清屏、设置当前颜色、颜色转换函数，这些函数在某些派生类中需要重写，它们被说明为虚函数。

• class VGAindirect　间接色彩类。该类新增了两个函数，用于写DAC色彩查找表和读DAC色彩查找表，该类说明并实现了这两个函数。该类没有实现任何由VGABASE类所说明的函数。

• class VGA16　16色模式类。该类实现了与色彩模式相关的函数在16色模式下的版本，并重写了置零清屏函数。该类说明并实现了两个新的函数：读调色板和写调色板。

• class VGA256　256色模式类。该类实现了与色彩模式相关的函数在256色模式下的版本。

• class VGAHIGH　高彩色模式类。该类实现了与色彩模式相关的函数在高彩色模式下的版本。

• class VGA32K　32K色模式类。该类重写了当前颜色设置函数和颜色转换函数。

• class VGA64K　64K色模式类。该类重写了当前颜色设置函数和颜色转换函数。

• class VGA16M　真彩色模式类。该类实现了与色彩模式相关的函数在真彩色模式下的版本。它说明并实现了用于真彩色模式的如下一些新的函数：象素点、扫描线、矩形块的亮度变化函数；叠加/去叠加写点、写扫描线、写矩形块函数。

• 图形模式类　共有18个图形模式类，每个类对应于一种图形模式，由这些类的名字很容易看出它们所对应的具体图形模式。这些类所完成的任务只有一项：在构造函数中进行参数设置。这些类处于类层次的最低层，它们都是可实例化的类，类体系中除这些类之外的其它类都是抽象类，都不能实例化。在应用程序中能直接使用的就是这18个图形模式类，其它的类只能用作派生新的图形模式类的基类。

该类体系中的各个类及其各个成员，将在后续的一些章节中陆续说明和实现它们：本章的后续部分将说明VGABASE类，并实现VGABASE类的几个基本成员函数；第4章至第7章，每章分别定义与一种色彩模式相关的若干个类，并实现所定义类的成员函数；第9章将实现VGABASE类中的各个绘图函数；第10章将对整套图形显示程序进行一个小结。

3. 基本图形类说明

系统程序3-2　VGABASE.H

```
--------------------------------------------------------------------------
typedef unsigned char unchar;

#define Select_Page(page) Select_Page_VESAint(page)
//or
//#define Select_Page(page) Select_Page_Trident(page)
//从所提供的两个换页函数中选择一个

#define G_SEGMENT 0a000h  //图形模式下视频地址空间的段址

class VGABASE {
```

```
public:
    int OK;  //图形模式初始化是否成功的标志
    COLOR CUR_COLOR;  //当前颜色值变量
    int WIDE,HIGH;  //当前图形模式的水平象素数，垂直象素数
    int VESAmodeNo;  //当前图形模式的VESA模式号或标准VGA模式号
    VGABASE() { OK=0; }
/*  以下5个函数为与图形模式完全无关的函数，它们在本类中得到实现  */
    virtual int init(void);  //图形模式初始化
    void close(void);  //关闭图形模式，返回文本模式
    void display_off(void);  //禁止屏幕刷新
    void display_on(void);  //允许屏幕刷新
    /*  以下5个函数虽然与色彩模式相关，但它们在本类中以常用方式得到实现。这些函数在某些派生类中需要重写，因此它们被说明为虚函数  */
        virtual void cls0(void);  //置零清屏
        virtual void setcolor(unchar color);  //设置当前颜色值
        virtual COLOR setcolorto(unchar color);  //颜色转换
    virtual void setcolor(unchar r,unchar g,unch0ar b);  //设置当前颜色值
    virtual COLOR setcolorto(unchar r,unchar g,unchar b);  //颜色转换
/*  以下8个函数为与色彩模式相关的函数，它们被说明为纯虚函数。本类没有实现这些函数，它们需要在各个色彩模式类中实现  */
    virtual void putpixel(int x,int y)=0;  //写点
    virtual void putpixel(int x,int y,COLOR color)=0;  //直接传递颜色写点
    virtual union COLOR getpixel(int x,int y)=0;  //读点
    virtual void scanline(int x1,int x2,int y)=0;  //画扫描线
    virtual void cls(void)=0;  //置色清屏
    virtual void getscanline(int x1,int y,int n,void *buf)=0;  //读扫描线
    virtual void putscanline(int x1,int y,int n,void *buf)=0;  //写扫描线
    virtual int scanlinesize(int x1,int x2)=0;
      //取保存一条扫描线所需缓冲区的大小
/*  以下25个函数为与图形模式无直接关系的函数，它们都在本类中得到实现。这些函数中的大部分都需要调用上面8个纯虚函数  */
    void line(int x1,int y1,int x2,int y2);  //画直线
    void moveto(int x,int y);  //置画线开始点的位置
    void lineto(int x,int y);  //在画线开始点与给定点间画一条直线
    void rectangle(int x1,int y1,int x2,int y2);  //画矩形
    void poly(int n,int *border);  //画多边形
    void poly(int *border);  //画多个多边形
    void circle(int x0,int y0,int r);  //画圆
    void sector(int x0,int y0,int r,int stangle,int endangle);  //画扇形
    void ellipse(int x0,int y0,long r1,long r2);  //画椭圆

    void setfillstyle(int fst);  //设置填充类型
    int getfillstyle(void);  //取当前的填充类型
    void setfillpattern(unsigned char *s);  //设置自定义填充类型
    void fillarea(int x0,int y0,COLOR bordercolor);  //填充任意区域
    void bar(int x1,int y1,int x2,int y2);  //画实矩形
    void polyfill(int n,int *border);  //画实多边形
    void polyfill(int *border);  //画有空洞的实多边形
```

```
    void circlefill(int x0,int y0,int r);  //画实圆
    void sectorfill(int x0,int y0,int r,int stangle,int endangle);  //画实扇形
    void ellipsefill(int x0,int y0,long r1,long r2);  //画实椭圆

    long imagesize(int x1,int y1,int x2,int y2);  //取保存块所需的缓冲区大小
    struct IMAGE *getimage(int x1,int y1,int x2,int y2,int where=inMEM);
      //读块到常规内存、扩展内存或硬盘
    void putimage(int x1,int y1,struct IMAGE *img);
      //写块自常规内存、扩展内存或硬盘
    void putimage(IMAGE *img);  //将块写到原处
    void getimageMEM(int x1,int y1,int x2,int y2,void *buf); //读块到常规内存
    void putimageMEM(int x1,int y1,int xn,int yn,void *buf); //写块自常规内存
    class XMS *getimageXMS(int x1,int y1,int x2,int y2);  //读块到扩展内存
    void putimageXMS(int x1,int y1,int xn,int yn,class XMS *xms);
      //写块自扩展内存
    char *getimageHD(int x1,int y1,int x2,int y2);  //读块到硬盘
    void putimageHD(int x1,int y1,int xn,int yn,char *filename);//写块自硬盘

protected:
/*  以下4个数据及3个函数是被保护的，在应用程序中它们是不可见的  */
    int CUR_X,CUR_Y;  //画线开始点的坐标
    int PAGEN;  //当前模式所占用的存储器页数
    int CUR_PAGE;  //当前存储器页
    int SCANLENG;  //当前模式下每条扫描线所占字节数

    virtual void setmode(void)  //模式设置
    void Select_Page_VESAint(int page);  //调用VESA BIOS功能的换页操作
    void Select_Page_Trident(int page);  //Trident的换页操作

public:
/*  以下18个函数为采用直接方式传递颜色值的图形显示函数，它们通过调用相应的采用间接传递方式的函数来实现，因此它们速度较慢。使用这些函数会隐含地改变当前颜色值。这些函数都以重载方式定义  */
    virtual void putpixel(int x,int y,COLOR color)=0;  //见程序说明
    virtual void scanline(int x1,int x2,int y,COLOR color)=0;  //见程序说明
    virtual void cls(COLOR color)=0;  //见程序说明
    void line(int x1,int y1,int x2,int y2,COLOR color)
        { CUR_COLOR=color; line(x1,y1,x2,y2); }
    void lineto(int x,int y,COLOR color)
        { CUR_COLOR=color; lineto(x,y); }
    void rectangle(int x1,int y1,int x2,int y2,COLOR color)
        { CUR_COLOR=color; rectangle(x1,y1,x2,y2); }
    void poly(int n,int *border,COLOR color)
        { CUR_COLOR=color; poly(n,border); }
    void poly(int *border,COLOR color)
        { CUR_COLOR=color; poly(border); }
    void circle(int x0,int y0,int r,COLOR color)
        { CUR_COLOR=color; circle(x0,y0,r); }
```

```
    void sector(int x0,int y0,int r,int stagl,int endagl,COLOR color)
        { CUR_COLOR=color; sector(x0,y0,r,stagl,endagl); }
    void ellipse(int x0,int y0,long r1,long r2,COLOR color)
        { CUR_COLOR=color; ellipse(x0,y0,r1,r2); }
    void fillarea(int x0,int y0,COLOR bordercolor,COLOR color)
        { CUR_COLOR=color; fillarea(x0,y0,bordercolor); }
    void bar(int x1,int y1,int x2,int y2,COLOR color)
        { CUR_COLOR=color; bar(x1,y1,x2,y2); }
    void polyfill(int n,int *border,COLOR color)
        { CUR_COLOR=color; polyfill(n,border); }
    void polyfill(int *border,COLOR color)
        { CUR_COLOR=color; polyfill(border); }
    void circlefill(int x0,int y0,int r,COLOR color)
        { CUR_COLOR=color; circlefill(x0,y0,r); }
    void sectorfill(int x0,int y0,int r,int stagl,int endagl,COLOR color)
        { CUR_COLOR=color; sectorfill(x0,y0,r,stagl,endagl); }
    void ellipsefill(int x0,int y0,long r1,long r2,COLOR color)
        { CUR_COLOR=color; ellipsefill(x0,y0,r1,r2); }
    };
```

程序说明：

当函数重载与继承同时使用时，那些具有相同函数名的函数必须在同一个类中实现，否则在较高层类中实现的函数将被屏蔽掉，而不能被应用程序使用。所以putpixel()、sanline()、cls()这三个函数的两个版本都必须放在各个色彩模式类中同时实现，而不能在这里首先实现它们的一个版本。

3.3.5 若干基本函数的实现

1. 图形初始化及关闭

图形初始化函数VGABASE::init()完成三项工作，一是，将保存当前颜色值的变量CUR_COLOR清零，这是一项必要的工作，因COLOR变量的长度为4个字节，各种色彩模式都没有全部使用这4个字节，而通常是用全部4个字节来进行颜色比较，这就需要将未用字节的值保持为0；二是，设置图形模式，但该函数并不直接完成这项工作，而是调用模式设置函数setmode()；三是，进行一次显示存储器换页操作，以使程序中所维持的显示存储器当前页号(CUR_PAGE)与实际相符。

通过将VGA的显示模式设置为3号字符模式来完成图形关闭工作，3号字符模式是DOS的缺省显示模式。

系统程序3-3 VGABASE.CPP

```
int VGABASE::init(void)
{
CUR_COLOR.dword=0L;
setmode();
if(PAGEN>1)
    Select_Page(0);
return(OK);  //OK在setmode()函数中确定
```

```
}

void VGABASE::setmode()
{
union REGS inregs;
OK=1;
if(VESAmodeNo>=0x100)  //如果是由VESA定义的扩展模式
    {
    inregs.x.bx=VESAmodeNo;
    inregs.x.ax=0x4f02;
    int86(0x10,&inregs,&inregs);  //调用VESA BIOS功能进行模式设置
    if(inregs.x.ax!=0x004f)  //设置不成功
        OK=0;
    }
else if(VESAmodeNo>=0)  //如果是标准VGA的模式
    {
    inregs.x.ax=VESAmodeNo;
    int86(0x10,&inregs,&inregs);  //调用标准VGA BIOS功能设置模式
    }
}

void VGABASE::close(void)
{
union REGS inregs;
inregs.x.ax=0x0003;
int86(0x10,&inregs,&inregs);
}
```

---

2. 显示存储器换页

实现了两个换页操作函数，一个用VESA BIOS功能5来实现，另一个用Trident VGA的寄存器操作来实现。只能从这两个函数中选择一个使用，通过一个宏替换来实现选择，该宏替换放在VGABASE.H中。对Trident VGA来说这分别是最慢和最快的两个换页函数，但它们之间的差别对图形显示速度的实际影响是极小的，因此通常应选择第一个函数，它具有最好的兼容性。还存在其它的换页操作方法，这里没有实现。

这两个函数除了完成换页操作，还要维持VGABASE::CUR_PAGE变量，该变量保存着显示存储器的当前页号，那些直接访问显示存储器的程序需要根据此变量来确定是否需要换页。

这两个函数都以所选页的页号作为输入参数，无返回参数。

系统程序3-4　VGABASE.CPP

```
void VGABASE::Select_Page_VESAint(int page)
{
asm {
    mov bx,0
    mov dx,page
    mov ax,4f05h
    int 10h
```

```
    }
CUR_PAGE=page;
}

void VGABASE::Select_Page_Trident(int page)
{
asm {
    mov bx,page
    xor bl,02h
    mov dx,3c4h
    mov al,0eh
    out dx,al
    inc dx
    mov al,bl
    out dx,al
    }
CUR_PAGE=page;
}
```

---

3. 屏幕软开关

即禁止/允许屏幕刷新。禁止屏幕刷新后，CPU能以最快的速度访问显示存储器，但这时屏幕图象将消失，屏幕变成全黑，好象显示器被关了一样。禁止屏幕刷新后，要用允许屏幕刷新操作来恢复屏幕显示。禁止屏幕刷新不会丢失显示存储器中的数据。

可通过标准VGA BIOS功能调用和VGA寄存器操作来实现这一功能，这里采用了BIOS功能调用。

系统程序3-5 VGABASE.CPP

```
void VGABASE::display_off(void)
{
asm {
    mov bl,36h
    mov ax,1201h
    int 10h
    }
}

void VGABASE::display_on(void)
{
asm {
    mov bl,36h
    mov ax,1200h
    int 10h
    }
}
```

---

4. 置零清屏

将所有象素点的颜色值全部置为零，这等同于将显示存储器中的每个字节都置为零，因此这一操作与各种色彩模式下象素点的数据格式无关，但在16色模式下必须辅以寄存器操作才能有效地访问显示存储器，因此这里所实现的函数不能用于16色模式，在16色模式类中该函数需要重写。为保证通用性和简洁性，该函数以页为单位对显示存储器清零，某些图形模式没有全部使用其所占用的每页显示存储器，这时，该函数就不是最快的。

系统程序3-6　VGABASE.CPP

```
--------------------------------------------------------------------------------
void VGABASE::cls0()
{
int pn=PAGEN;  //当前显示模式所占用的存储器总页数
display_off();  //禁止刷新
asm mov si,0
loop:
Select_Page(_SI);
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0
    mov ax,0
    mov cx,8000h
    cld
    rep stosw
    inc si
    cmp si,pn
    jne loop
    }
display_on();  //恢复刷新
}
--------------------------------------------------------------------------------
```

5. 颜色转换及当前颜色设置

这组函数用于将各色彩模式下的原始颜色值转换、设置为由COLOR联合所表示的颜色值。在间接色彩模式下，原始颜色值为1个字节的颜色索引值，在直接色彩模式下，原始颜色值为3个字节的红绿蓝三基色的亮度值，这里实现了对这两种原始颜色值进行转换和设置的函数，但在每种色彩模式下只存在一种原始颜色值。

系统程序3-7　VGABASE.CPP

```
--------------------------------------------------------------------------------
/*  只在间接色彩模式下有用  */
void VGABASE::setcolor(unchar col)
{
CUR_COLOR.byte=col;
}

/*  只在真彩色模式下有用，在15位色和16位色模式下需重写  */
void VGABASE::setcolor(unchar r,unchar g,unchar b)
{
CUR_COLOR.rgb[0]=b;
```

```
CUR_COLOR.rgb[1]=g;
CUR_COLOR.rgb[2]=r;
}

/*  只在间接色彩模式下有用  */
COLOR VGABASE::setcolorto(unchar col)
{
COLOR A={0};
A.byte=col;
return(A);
}

/*  只在真彩色模式下有用，在15位色和16位色模式下需重写  */
COLOR VGABASE::setcolorto(unchar r,unchar g,unchar b)
{
COLOR A={0};
A.rgb[0]=b;
A.rgb[1]=g;
A.rgb[2]=r;
return A;
}
```

第 4 章　256色模式的图形操作

4.1　概　　述

程序将提供对五种256色图形模式的支持，它们是：320×200×256色、640×400×256色、640×480×256色、800×600×256色、1024×768×256色。

由表2-1可见，这五种图形模式包含了显示存储器分页的所有三种情况，即不分页、行外分页、行内分页，这里将针对最不利的一种情况——行内分页进行编程，在不分页和行外分页模式中，该程序运行不是最快的，但其与最快运行程序在速度上差别极小。

256色模式下的显示存储器结构参见2.1节中的有关内容，在该模式下访问显示存储器不需要对任何VGA寄存器进行操作，也不需要对象素颜色值进行任何变换，与其它的图形模式相比，256色模式下的图形操作是最为直接、最为简单的。

下面将结合具体的编程来介绍256色模式下的图形操作技术。

在各种图形模式及色彩模式下，很多编程上的处理都是相同或类似的。本章将对图形操作编程中的一些具体处理作出较详细的说明，以后各章中遇到类似问题都将不再说明。因此建议较详细地阅读本章中的程序。

4.2　编 程 方 案

这里定义了与256色模式相关的7个类：VGAindirect、VGA256、_320_200_256、_640_400_256、_640_480_256、_800_600_256、_1024_768_256，对这7个类的定义及它们之间的继承关系即构成了256色模式下图形操作的编程方案。这7个类的继承关系见图3-1。系统程序4-1、系统程序4-2给出了这7个类的说明。

系统程序4-1　VGABASE.H

```
--------------------------------------------------------------------------
/*  间接色彩模式类  */
class VGAindirect : public VGABASE  //派生于VGABASE类
    {
public:
    void setdac(unchar idx,unchar r,unchar g,unchar b);  //置DAC色彩查找表
    void getdac(unchar idx,unchar &r,unchar &g,unchar &b);  //取DAC色彩查找表
    } ;
--------------------------------------------------------------------------
```

系统程序4-2　VGA256.H

```
--------------------------------------------------------------------------
/*  256色模式类  */
class VGA256 : public VGAindirect
    {
public:
/*  以下重新说明了VGABASE类中的10个图形操作函数，这些函数在VGABASE类中都是纯虚函数  */
    void putpixel(int x,int y);
```

```
    union COLOR getpixel(int x,int y);
    void cls(void);
    int scanlinesize(int x1,int x2);
    void scanline(int x1,int x2,int y);
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);

    void putpixel(int x,int y,COLOR color)
        { CUR_COLOR=color; putpixel(x,y); }
    void scanline(int x1,int x2,int y,COLOR color)
        { CUR_COLOR=color; scanline(x1,x2,y); }
    void cls(COLOR color)
        { CUR_COLOR=color; cls(); }
    };

/*  320×200×256色模式类  */
class _320_200_256 : public VGA256
    {
public:
    _320_200_256();  //构造函数，进行参数设置
    };

/*  640×400×256色模式类  */
class _640_400_256 : public VGA256
    {
public:
    _640_400_256();
    };

/*  640×480×256色模式类  */
class _640_480_256 : public VGA256
    {
public:
    _640_480_256();
    };

/*  800×600×256色模式类  */
class _800_600_256 : public VGA256
    {
public:
    _800_600_256();
    };

/*  1024×768×256色模式类  */
class _1024_768_256 : public VGA256
    {
public:
    _1024_768_256();
```

```
    };
-------------------------------------------------------------------------
```

4.3　点　操　作

4.3.1　操作步骤

点操作包括写点、读点两项操作，在程序中这两项操作的通常步骤如下：

①计算所操作点的地址，包括页号和偏移量，计算方式如下：

Page＝(Width*Y＋X)/10000H

Offset＝(Width*Y＋X)%10000H

Page——象素点所在的页号

Offset——象素点在页中的偏移量

Width——水平象素数，在256色模式下，它就等于一条扫描线所占的字节数

X、Y——象素点的坐标位置

在汇编语言中，能够很简捷地完成地址计算。

②比较象素点所在页号是否与显示存储器的当前页号相同，如不同则进行换页操作。

③在内存地址A0000H:Offset处写入或读出1个字节，该字节为象素点的颜色值。

4.3.2　程序

系统程序4-3　256.CPP

```
-------------------------------------------------------------------------
/************************************************************************
功能：写一个象素点
输入参数：  x＝所写象素点在屏幕上的横向坐标值
            y＝所写象素点在屏幕上的纵向坐标值
返回值：无
    直接传递颜色值的写点函数，在VGA256类说明中直接以内部函数的形式进行了定义，它通过调用
本函数实现。
************************************************************************/
void VGA256::putpixel(int x,int y)
{
unchar color=CUR_COLOR.byte;  //象素点的颜色值
int scanleng=SCANLENG;  //每条扫描线所占字节数
int cur_page=CUR_PAGE;  //显示存储器的当前页
asm {
    mov ax,y
    mul scanleng  //16位的乘法运算，进位将自动放在dx中
    add ax,x  //加上象素点的x坐标值
/*  如果是不分页或行外分页的模式，可去掉以下5行  */
    jnc jemp1  //判断是否有进位
    inc dx  //如有进位将dx加1
    }
jemp1:
asm {
```

```
    mov di,ax  //将ax中的地址偏移量放到di中，见程序说明
/*  如果是不分页的模式，可以去掉以下6行  */
    cmp dx,cur_page  //将dx中所存放的页号与显示存储器的当前页号进行比较
    je jemp2  //如果相同转到jemp2，如果不同则执行如下的换页操作
    }
    Select_Page(_DX);  //换页操作，见程序说明
jemp2:
asm {
    mov ax,G_SEGMENT  //G_SEGMENT为视频地址空间的段址，其在程序3-2中定义
    mov es,ax  //将视频地址空间的段址放到es中
    mov al,color
    mov es:[di],al  //写显示存储器
    }
}

/*************************************************************************
功能：读一个象素点
输入参数：  x＝所读象素点在屏幕上的横向坐标值
            y＝所读象素点在屏幕上的纵向坐标值
返回值：COLOR＝所读象素点的颜色值
*************************************************************************/
union COLOR VGA256::getpixel(int x,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
union COLOR color={0L};
asm {
    mov ax,y
    mul scanleng
    add ax,x
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov al,es:[di]  //读显示存储器
    }
color.byte=_AL;
return(color);
```

```
}
-------------------------------------------------------------------------
```

程序说明：

在进行C、C++的函数调用时，_DS、_BP、_DI、_SI这四个寄存器的值将自动得到保存，因而程序中保存在_DI中的地址偏移量，在调用了换页操作函数后仍然保持有效。

4.4　扫描线操作

4.4.1　操作步骤

扫描线操作包括画扫描线、读扫描线、写扫描线，还包括一个辅助操作用于取保存扫描线所需缓冲区的大小。进行扫描线操作时，需要指定两个端点的横坐标值x1、x2及它们共同的纵坐标值y，扫描线操作的一般步骤如下：

①对两个端点的横坐标值进行比较和交换，使x1≤x2；计算出扫描线的长度(减1)：leng＝x2－x1；

②计算端点(x1,y)的页号Page1及偏移量Offset1；

③将Page1与显示存储器的当前页号进行比较，若不等则进行换页操作；

④若Offset1＋leng>FFFFH，则表示扫描线跨页，转到⑥；

⑤将leng加1；在从A000H:Offset1开始长度为leng的地址区域上进行画、读或写操作；结束。

⑥在A000H:Offset1至A0000H:FFFFH的地址区域上进行画、读或写操作，完成上页中的扫描线操作；

⑦将显示存储器换到下一页；

⑧计算扫描线在下页中的长度：leng2＝Offset1＋leng－FFFFH；在从A0000H:0000H开始长度为leng2的地址区域进行画、读或写操作，完成下页中的扫描线操作；结束。

4.4.2　程序

系统程序4-4　256.CPP

```
-------------------------------------------------------------------------
/*************************************************************************
功能：以当前颜色在屏幕上画一条水平直线
输入参数：  x1＝扫描线始点的横坐标
            x2＝扫描线末点的横坐标
            y ＝扫描线的纵坐标
返回值：无
    直接传递颜色值的画扫描线函数，在VGA256类说明中直接以内部函数的形式进行了定义，它通过调用本函数实现。
*************************************************************************/
void VGA256::scanline(int x1,int x2,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unchar color=CUR_COLOR.byte;
```

```
asm {
/*  以下7行，使x1≤x2，并计算扫描线长度(减1)，结果放在ax中*/
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
    mov x1,bx
    }
jemp1:
asm {
    push ax  //将扫描线长度(减1)入栈
    mov ax,y
    mul scanleng
    add ax,x1
    jnc jemp2
    inc dx
    }
jemp2:
asm {
    mov di,ax  //至此，计算得到扫描线始点的地址，页号在dx中,偏移量在di中
    cmp dx,cur_page
    je jemp3
    mov cur_page,dx
    }
    Select_Page(cur_page);
jemp3:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov ax,di
    pop cx  //扫描线长度(减1)出栈，放在cx中
/*  若是不换页或行外换页的模式，以下2行语句可以去掉  */
    add ax,cx
    jc jemp4  //判断扫描线是否跨页，如跨页转到jemp4
/*  以下4行，完成扫描线不跨页时的操作  */
    inc cx
    mov al,color
    cld
    rep stosb  //带重复前缀的存串指令，一次画完整个扫描线
    jmp end  //结束
    }
jemp4:
/*  以下为扫描线跨页时的操作，若是不换页或行外换页的模式，可以去掉它们  */
asm {
    push ax  //扫描线在下页中的长度(减1)，入栈
```

```
    sub cx,ax
    mov al,color
    cld
    rep stosb  //画出上页的扫描线
    inc cur_page
    }
    Select_Page(cur_page);  //将显示存储器换到下一页
asm {
    mov ax,G_SEGMENT
    mov es,ax
    pop cx  //扫描线在下页中的长度(减1)，出栈
    inc cx
    mov di,0h
    mov al,color
    cld
    rep stosb  //画出下页的扫描线
    }
end:
}

/*************************************************************************
功能：读出一条扫描线上所有点的颜色值
输入参数：x1＝扫描线始点的横坐标
          y＝扫描线的纵坐标
          n＝扫描线的长度
          buf＝保存扫描线的缓冲区的地址
返回值：无
        扫描线上所有点的颜色值顺序存于buf中
*************************************************************************/
void VGA256::getscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
int k1,k2;
asm {
/*  以下14行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov si,ax
    cmp dx,cur_page
    je jemp2
    mov cur_page,dx
```

```
    }
    Select_Page(cur_page);
jemp2:
asm {
    mov ax,si
    mov cx,n
    dec cx
    add ax,cx
    jc jemp3  //扫描线跨页，转到jemp3
    push ds  //见程序说明
/*  以下6行，为扫描线不跨页时的操作*/
    inc cx
    les di,buf
    mov ax,G_SEGMENT
    mov ds,ax
    cld
    rep movsb
    pop ds  //见程序说明
    jmp end
    }
jemp3:
/*  以下为扫描线跨页时的操作  */
asm {
    push ds  //见程序说明
    mov k1,ax
    sub cx,ax
    les di,buf
    mov ax,G_SEGMENT
    mov ds,ax
    cld
    rep movsb  //读出上页的扫描线
    mov k2,di
    pop ds  //见程序说明
    inc cur_page
    }
    Select_Page(cur_page);  //将显示存储器换到下页
asm {
    push ds  //见程序说明
    mov cx,k1
    inc cx
    les di,buf
    mov di,k2
    mov ax,G_SEGMENT
    mov ds,ax
    mov si,0h
    cld
    rep movsb  //读出下页的扫描线
    pop ds  //见程序说明
```

```
    }
end:;
}

/*************************************************************************
功能：将由getscanline()所读出的扫描线写到屏幕上
输入参数：x1＝扫描线始点的横坐标
          y＝扫描线的纵坐标
          n＝扫描线的长度
          buf＝保存扫描线的缓冲区的地址
返回值：无
*************************************************************************/
void VGA256::putscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
int k1,k2;
asm {
/*  以下14行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov di,ax
    cmp dx,cur_page
    je jemp2
    mov cur_page,dx
    }
    Select_Page(cur_page);
jemp2:
asm {
    mov ax,di
    mov cx,n
    dec cx
    add ax,cx
    jc jemp3  //若扫描线跨页，转到jemp3
    push ds
    inc cx
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf
    cld
    rep movsb  //扫描线不跨页时，写入整条扫描线
    pop ds
```

```
    jmp end
    }
jemp3:
/*  以下为扫描线跨页时的操作  */
asm {
    push ds
    mov k1,ax
    sub cx,ax
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf
    cld
    rep movsb  //写入上页的扫描线
    mov k2,si
    pop ds
    inc cur_page
    }
    Select_Page(cur_page);  //显示存储器换到下页
asm {
    push ds
    mov cx,k1
    inc cx
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0h
    lds si,buf  //写入下页的扫描线
    mov si,k2
    cld
    rep movsb
    pop ds
    }
end:;
}

/*************************************************************************
功能：取保存一条扫描线所需缓冲区的大小，以字节计
输入参数：  x1＝扫描线始点的横坐标
            x2＝扫描线末点的横坐标
返回值：int＝所需缓冲区的大小
*************************************************************************/
int VGA256::scanlinesize(int x1,int x2)
{
int k;
if(x1>x2)
    { k=x1; x1=x2; x2=k; }
k=x2-x1+1;
if( k&0x0001 )  //如果长度为奇数
    k++;  //使长度变为偶数，加快扩展内存的操作
```

```
return(k);
}
```

程序说明：

在嵌入汇编程序中，应尽量不要使用CS、DS、SS、SP、BP这几个寄存器，但在很多情况下DS寄存器的使用往往是不可避免的，如果使用了这些寄存器，那就应该在使用之前保存它们的状态，用完之后恢复它们的状态。在嵌入汇编中ES、DI、SI这三个寄存器的状态是无需保存的。

4.5　清　　屏

256色模式的置零清屏函数在VGABASE类中实现，这里只定义置色清屏函数。为了使函数更加简洁，这一函数以页为单位进行清屏，此显示模式所占用的最后一页没有全部使用，这时该函数会多写一些显示存储器，这会使速度稍慢。本函数有两个版本，这里只定义间接传递颜色值的版本，直接传递颜色值的版本在VGA256类说明中以内部函数的形式定义。

系统程序4-5　256.CPP

```
/*  256色模式下置色清屏  */
void VGA256::cls()
{
unchar col=CUR_COLOR.byte;
int pn=PAGEN;  //当前模式所占用的显示存储器页数

asm mov si,0
loop:
Select_Page(_SI);
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0h
    mov ah,col
    mov al,col
    mov cx,8000h
    cld
    rep stosw  //带重复前缀的字存串指令，一次写一页

    inc si
    cmp si,pn
    jne loop
    }
}
```

4.6 DAC色彩查找表

4.6.1 原理及操作技术

256色模式是一种间接色彩模式，在该模式下，显示存储器中存放的是每个象素点的颜色索引值，DAC色彩查找表确定了每个颜色索引值所对应的真实颜色值。这里实现了对DAC色彩查找表进行设置和读取的两个函数，这两个函数定义在VGAindirect类中，它们在16色模式下也可以使用。

DAC色彩查找表存放在一组DAC色彩寄存器中，可通过标准VGA BIOS功能调用和VGA寄存器操作来读、写DAC色彩查找表，这里采用了寄存器操作。对DAC色彩寄存器进行写操作的步骤为：首先将一个8位的颜色索引值写入3C8h端口；然后对3C9h端口连续进行三次写操作，依次写入红绿蓝三基色的亮度值。对DAC色彩寄存器进行读操作的步骤为：首先将8位的颜色索引值写入3C7h端口；然后对3C9h端口连续进行三次读操作，依次读出红绿蓝三基色的亮度值。每次读或写的基色亮度值数据都只有低6位有效。

DAC色彩查找表的详细原理及操作技术可参见1.3.1、1.3.3、2.2.7节中的有关内容。

在DAC色彩查找表中，每个基色使用6个数据位，可具有64级亮度，而在应用程序中往往更习惯于用8位数据即256级亮度来控制基色的亮度，因此函数输入输出的基色值都转换为8位，即0为最暗，255为最亮。

4.6.2 程序

系统程序4-6 VGABASE.CPP

```
--------------------------------------------------------------------------------
/*************************************************************************
功能：设置某个颜色索引值所对应的真实颜色值
输入参数：idx＝颜色索引值(0～255)
          red＝真实颜色值中红色的亮度(0～255)
          green＝真实颜色值中绿色的亮度(0～255)
          blue＝真实颜色值中蓝色的亮度(0～255)
返回值：无
*************************************************************************/
void VGAindirect::setdac(unchar idx,unchar red,unchar green,unchar blue)
{
asm {
    mov dx,3C8h
    mov al,idx
    out dx,al

    mov cl,2
    inc dx  //将I/O端口地址转换为3C9h
    mov al,red
    shr al,cl  //将8位的基色值转换为6位
    out dx,al
    mov al,green
    shr al,cl
```

```
    out dx,al
    mov al,blue
    shr al,cl
    out dx,al
    }
}

/*************************************************************************
功能：取某个颜色索引值所对应的真实颜色值
输入参数：idx＝颜色索引值
          red＝存放所返回的真实颜色值的红色亮度
          green＝存放所返回真实颜色值的绿色亮度
          blue＝存放所返回真实颜色值的蓝色亮度
返回值：无
        返回的真实颜色值存放在通过引用传递的red、green、blue三个参数之中。
*************************************************************************/
void VGAindirect::getdac(unchar idx,unchar &red,unchar &green,unchar &blue)
{
unchar r,g,b;
asm {
    mov dx,3C7h
    mov al,idx
    out dx,al

    mov cl,2
    mov dx,3C9h
    in al,dx
    shl al,cl
    mov r,al
    in al,dx
    shl al,cl
    mov g,al
    in al,dx
    shl al,cl
    mov b,al
    }
red=r; green=g; blue=b;
}
```

4.6.3　应用

与那些直接色彩模式相比，间接色彩模式的色彩表现能力是较差的，但间接色彩模式也具有一个很好的特性，就是可以通过色彩查找表来直接改变象素的颜色，而不需要重写显示存储器，利用这一特性可以方便、快速地实现淡入淡出及一些特殊的动画效果。

1. 淡入淡出

淡入淡出包含淡入和淡出两种特殊的显示效果，淡入是指，一幅图象缓缓地从背景中出现，亮度逐渐增强直至稳定显示在屏幕上；淡出是指，一幅图象的亮度逐渐减弱，直至完全消失在背景之中。

实现淡入的基本方法为：图象和背景采用不同的颜色索引值，在画图象之前将图象的真实颜色值设置为与背景相同，画出图象之后，再按一定的过程逐渐地增强图象颜色索引值所对应的真实颜色值，直至达到一种稳定的真实颜色值。

实现淡出的基本方法为：按一定的过程逐渐减弱图象颜色索引值所对应的真实颜色值，直至与背景颜色值完全相同，这时图象就消失在背景之中。

下面给出实现淡入淡出的一个简单的示例程序。该程序实现一个实矩形在屏幕上的淡入淡出。

示例程序4-1

```
void main()
{
_640_480_256 A;
A.init();
A.cls0();
A.setcolor(1);
A.setdac(1,0,0,0);
A.bar(100,100,200,200);
/*  以下循环实现淡入过程  */
for(int i=1;i<256;i++)
    {
    delay(20);
    A.setdac(1,i,i,i);
    }
getch();
/*  以下循环实现淡出过程  */
for(int i=255;i>=0;i--)
    {
    delay(20);
    A.setdac(1,i,i,i);
    }
getch();
A.close();
}
```

通过细心的设计可以实现具有多种颜色的图象在较复杂背景中的淡入淡出。

2. 特殊动画

修改色彩查找表不能作为一般的动画显示方法，但它可用来简捷地实现一些特殊的动画效果，如流动、闪烁、下雨、下雪、闪电、瀑布等。下面给出一个反映液体在输送管中流动情况的示例性程序。

示例程序4-2

```
void main()
{
```

```
_320_200_256 A;
A.init();
A.cls0();
A.setcolor(7);
A.line(0,100,319,100);  //画出上管壁
A.line(0,104,319,104);  //画出下管壁
int i,k,col[]={255,191,127,255,191,127};
for(i=0;i<3;i++)
    A.setdac(i+1,col[i],0,0);  //设置液体颜色
k=1;
/*  以下循环画出管中液体，使用的颜色索引值为1、2、3  */
for(i=0;i<320;i++)
    {
    A.setcolor(k);
    A.line(i,101,i,103);
    k++;
    if(k==4) k=1;
    }
k=0;
/*  在以下循环中实现流动效果  */
do {
    for(i=0;i<3;i++)
        A.setdac(i+1,col[k+i]);
    k++;
    if(k==3) k=0;
    delay(300);
    } while(keybios(1));
A.close();
}
```

4.7　参 数 设 置

参数设置是需要在各个图形模式类中定义的唯一一个函数，那些属于同一色彩模式的各个图形模式类，只在这个函数上存在差别。参数设置由构造函数来完成，所需设置的参数包括：屏幕水平象素数、屏幕垂直象素数、每条扫描线所占字节数、当前显示模式所占用的存储器页数、VESA模式号或标准VGA模式号。

共有5个256色图形模式类，每个类构造函数及模式设置函数如下。

系统程序4-7　256.CPP

```
--------------------------------------------------------------------------------
/*  320×200×256色参数设置*/
320_200_256::_320_200_256()
{
SCANLENG=320;  //每条扫描线所占字节数，在256色模式下，其等于水平象素数
WIDE=320;  //屏幕水平象素数
HIGH=200;  //屏幕垂直象素数
PAGEN=1;  //所占用的存储器页数
VESAmodeNo=0x0013;  //模式号，这是标准VGA模式号
}
```

```
/*  640×400×256色参数设置  */
_640_400_256::640_400_256()
{
SCANLENG=640;
WIDE=640;
HIGH=400;
PAGEN=4;
VESAmodeNo=0x0100;  //这是VESA模式号
}

/*  640×480×256色参数设置  */
_640_480_256::_640_480_256()
{
SCANLENG=640;
WIDE=640;
HIGH=480;
PAGEN=5;
VESAmodeNo=0x0101;
}

/*  800×600×256色参数设置  */
_800_600_256::_800_600_256()
{
SCANLENG=800;
WIDE=800;
HIGH=600;
PAGEN=8;
VESAmodeNo=0x0103;
}

/*  1024×768×256色参数设置  */
_1024_768_256::_1024_768_256()
{
SCANLENG=1024;
WIDE=1024;
HIGH=768;
PAGEN=12;
VESAmodeNo=0x0105;
}
--------------------------------------------------------------------------------
```

第 5 章　16色模式的图形操作

5.1　概　　述

程序将实现对三种16色图形模式的支持，它们是：640×480×16色、800×600×16色、1024×768×16色。由表2-1可见，这三种图形模式包含了两种显示存储器分页情况：不分页和行外分页，这里将针对行外分页情况进行编程，在那些行内分页的16色模式中本程序将不能使用，如1280×1024×16色、1600×1200×16色，但目前的普通VGA系统都不支持这些模式。

16色模式下的显示存储器结构参见2.1节中的有关内容。该模式下对显示存储器的访问要涉及到较广泛的寄存器操作，有关的寄存器操作参见2.2.4节和2.2.5节。由于采用了位面技术且需要进行寄存器操作，因而，与其它的图形模式相比，16色模式下的图形操作是最为复杂的。

16色模式下增加了两个调色板操作函数。

下面首先介绍16色模式下两个较特殊的问题。

5.1.1　地址计算

由于采用了位面技术，16色模式下的每个象素点在每个位面中只占1位，因此地址计算中除了要算出象素点所在的页号和字节偏移量之外，还需算出象素在字节中所处的位，并确定出相应的屏蔽码。具体计算方式如下：

```
Page=(Width*Y+X)/8/10000H
Offset=[(Width*Y+X)/8]%10000H
Bit=X%8
Mask=80H>>Bit
```

Page——象素点所在的页号

Offset——象素点在页中偏移量

Bit——象素点在所在字节中所处的位，0表示处于位7，7表示处于位0

Mask——所在字节的屏蔽码

Width——水平象素数

X、Y——象素点的坐标位置

在16色模式下水平象素数除以8等于一条扫描线在一个位面中所占字节数。

5.1.2　寄存器操作策略

在16色模式下，对显示存储器进行访问时，需要有关的VGA寄存器保持有正确的状态，访问显示存储器的方式有多种，每种方式所要求的寄存器状态都存在差别，而在很多情况下寄存器操作往往比访问显示存储器更加费时，这就需要安排一定的策略来尽量减少寄存器操作的次数，并保证操作的正确性，可采用的策略有以下三种：

(1)最安全策略

在进行一项图形操作之前，保存这次操作所要改变的所有寄存器的状态，完成操作后再恢复这些寄存器的状态。这是一种最保险的策略，但其速度也是最慢的。这一策略通常为中断程序和调试程序所采用。

(2)最快策略

针对在速度上受寄存器操作影响最大且最为常用的一种操作，这往往是写点操作，给有关寄存器规定一个缺省状态，当一项操作所需的寄存器状态为缺省状态时，就不需对寄存器进行操作，当一项操作改变了寄存器的缺省状态时就在退出前恢复它。这种策略能获得最快的速度，但它不安全，因为系统中的视频BIOS字符及图形输出功能、鼠标之类的设备驱动程序，在完成它们的功能时都会改变VGA寄存器的状态，而且很少恢复它们，也更不会遵守你所给出的缺省约定。因此只有在能保证不使用第二类图形操作程序的情况下，才能使用这一策略。

(3)折中策略

在操作中，对所有必须的寄存器的状态进行设置，而不依赖于任何缺省约定，但也不保存和恢复任何寄存器的状态。这一策略在安全和速度间进行了折中，它具有较高的安全性也能获得较快的速度。当系统中存在采用最快策略的程序时，这一策略就会产生错误，但很少有那个程序会采用最快策略。

在本书的编程中将采用折中策略。

5.2 编 程 方 案

这里定义了与16色模式相关的4个类，它们是：VGA16、_640_480_16、_800_600_16、_1024_768_16，另有一个相关的类VGAindirect已在5.2中定义，这些类的继承关系见图3-1。下面给出这4个类的说明。

系统程序5-1 VGA16.H

```
--------------------------------------------------------------------------
/*  16色模式类  */
class VGA16 : public VGAindirect
    {
public:
    void putpixel(int x,int y);
    union COLOR getpixel(int x,int y);
    void cls(void);
    void cls0(void);
    int scanlinesize(int x1,int x2);
    void scanline(int x1,int x2,int y);
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);

    void putpixel(int x,int y,COLOR color)
        { CUR_COLOR=color; putpixel(x,y); }
    void scanline(int x1,int x2,int y,COLOR color)
```

```
        { CUR_COLOR=color; scanline(x1,x2,y); }
    void cls(COLOR color)
        { CUR_COLOR=color; cls(); }

    void setpalette(unchar idx16,unchar idx64);  //置调色板
    unchar getpalette(unchar idx16);  //取调色板
    };

/*  640×480×16色模式类  */
class _640_480_16 : public VGA16
    {
public:
    _640_480_16();
    };

/*  800×600×16色模式类  */
class _800_600_16 : public VGA16
    {
public:
    _800_600_16();
    };

/*  1024×768×16色模式类  */
class _1024_768_16 : public VGA16
    {
public:
    _1024_768_16();
    };
-------------------------------------------------------------------------
```

5.3　写　　点

5.3.1　操作步骤

在16色模式下，VGA提供了4种写显示存储器的方式，写点操作通常只使用方式0或方式2，下面按折中策略分别介绍这两种写方式的操作步骤。

1. 写方式0

对写方式0发生影响的寄存器有：定序器中的映象屏蔽寄存器、图形控制器中的位屏蔽寄存器、设置重置寄存器、设置重置允许寄存器、循环移位寄存器。

在该方式下，通过映象屏蔽寄存器确定象素点的颜色，以0FFh写显示存储器。为了不使写入的颜色与原有颜色叠加，在写入之前必须将象素点的原有颜色清零。

在该方式下，为了使位屏蔽寄存器正确发挥作用，在写显示存储器之前，必须在相同地址处进行一次存储器读操作。

该方式的操作步骤如下：

①计算象素点作在的页及偏移量，必要时进行换页操作；

②写图形控制器中的模式寄存器，将写方式置为写方式0(可作为缺省状态)；

③写设置重置允许寄存器，禁止所有位面的设置重置(可作为缺省状态)；

④写循环移位寄存器，不作循环移位和逻辑运算(可作为缺省状态)；

⑤计算对象素点所在字节的屏蔽码，并用该屏蔽码写位屏蔽寄存器；

⑥将象素点原有颜色清零：写映象屏蔽寄存器，使所有位面写允许；在象素点所在字节，进行一次存储器读操作，以装载锁存器，使位屏蔽发生作用；以0写存储器。

⑦以颜色值写映象屏蔽寄存器；

⑧进行一次存储器读操作；以0FFh写存储器。

定序器中的映象屏蔽寄存器使写入的数据对各位面有效或无效；位屏蔽寄存器使写入的1个字节的各位有效或无效；设置重置允许寄存器确定了那些有效位面及有效位的数据是来自于CPU还是来自于设置重置寄存器；当循环移位寄存器被置位时，来自CPU的数据将执行循环移位或逻辑运算。这是一种最灵活的写方式，但在很多情况下它的速度不如其它方式。

2. 写方式2

对写方式2发生影响的寄存器有：定序器中的映象屏蔽寄存器、图形控制器中的位屏蔽寄存器。

在该方式下，为了使位屏蔽寄存器正确发挥作用，在写显示存储器之前，必须在相同地址处进行一次存储器读操作。

该方式的具体操作步骤如下：

①计算象素点所在的页及偏移量，必要是进行换页操作；

②写映象屏蔽寄存器，使四个位面全部写允许；(可作为缺省状态)

③写图形控制器中的模式寄存器，选择写方式2；

④计算屏蔽码，并用该码写位屏蔽寄存器；

⑤在地址处读显示存储器；在地址处以颜色值写显示存储器。

由以上的操作步骤可以看出，在折中策略下，写方式2的速度明显较快，即使采用最快策略，写方式2具有较快的速度，因为写方式0需要进行两次存储器读写操作。本书采用写方式2实现写点函数。

5.3.2 程序

系统程序5-2 16.CPP

```
--------------------------------------------------------------------------------
#define GRAPHICS_CTL 3ceh
  //图形控制器寄存器组索引端口地址，其数据端口地址为该值加1
#define SEQUENCE_CTL 3c4h
  //定序器寄存器组索引端口地址，其数据端口地址为该值加1

/*  16色模式下写一个象素点 */
void VGA16::putpixel(int x, int y)
{
unchar color=CUR_COLOR.byte;  //颜色值，仅低4位有效
int scanleng=SCANLENG;  //每条扫描线在一个位面中所占字节数
int cur_page=CUR_PAGE;  //显示存储器的当前页
asm {
```

```
/*  以下11行，计算偏移地址(DI)及页(DX)，并在必要时完成换页操作  */
    mov bx,x
    mov cl,3
    shr bx,cl  //bx除以8
    mov ax,y
    mul scanleng
    add ax,bx
    mov di,ax
    cmp dx,cur_page
    je jemp1:
    }
    Select_Page(_DX);
jemp1:
asm {
/*  以下6行，设置定序器中的映象屏蔽寄存器  */
    mov dx,SEQUENCE_CTL  //定序器索引端口地址3C4h
    mov al,2
    out dx,al
    inc dx
    mov al,0fh  //四个位面全部写允许
    out dx,al
/*  以下6行，选择写方式2  */
    mov dx,GRAPHICS_CTL  //图形控制器索引端口地址3CEh
    mov al,05h
    out dx,al
    inc dx
    mov al,02h
    out dx,al
/*  以下9行，计算屏蔽码，并设置位屏蔽寄存器  */
    dec dx
    mov al,08h
    out dx,al
    mov cx,x
    and cl,7
    mov al,80h
    shr al,cl
    inc dx
    out dx,al

    mov ax,G_SEGMENT
    mov es,ax
    mov al,color
    mov ah,es:[di]
      //写之前进行一次读操作，以装载锁存器，使位屏蔽正确发生作用
    mov es:[di],al  //写显示存储器
    }
}
-------------------------------------------------------------------------
```

5.4 读　　点

5.4.1 操作步骤

在16色模式下，VGA提供了两种读显示存储器的方式，一般的读点操作通常采用读方式0。在读方式0中，每次读操作只能读出象素点在一个位面中的值，因此需要进行4次读操作，以分别读出象素点在各个位面中的值。读方式0的操作步骤为：

①计算地址，包括页号、偏移量及屏蔽码，完成必要的换页操作；

②写图形控制器的模式寄存器，选择读方式0；

③进行一个4次的循环，在每次循环中完成如下工作：

写读位面选择寄存器，依次选择要读的位面；

读显示存储器，得到当前位面中8个象素点的颜色值；

用屏蔽码确定单个象素点在当前位面中的颜色值；

将单个位面的颜色值逐步合成位完整的颜色值。

④返回所读象素点的颜色值。

5.4.2 程序

系统程序5-3　16.CPP

```
--------------------------------------------------------------------------
/*  16色模式下读一个象素点  */
COLOR VGA16::getpixel(int x, int y)
{
COLOR color={0};
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下12行，计算页号及偏移量，必要时进行换页操作  */
    mov bx,x
    shr bx,1
    shr bx,1
    shr bx,1
    mov ax,y
    mul scanleng
    add ax,bx
    mov di,ax
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
jemp1:
asm {
/*  以下4行，计算屏蔽码，放于bl中  */
    mov cx,x
    and cl,7
    mov bl,80h
```

```
    shr bl,cl
/*  以下6行，选择读方式0  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
    mov al,0
    out dx,al

    mov ax,G_SEGMENT
    mov es,ax
/*  以下4行，将寄存器操作指向读位面选择寄存器  */
    dec dx
    mov al,04h
    out dx,al
    inc dx
    mov al,3  //al存放所要读的位面，在下面的循环中它将逐步递减
    mov ch,0  //ch用于存放所读的颜色值，在下面的循环中它将逐步置位
    }
loop1:  //相对于4个位面的循环
asm {
    out dx,al  //选择所读的位面
    shl ch,1
    mov ah,es:[di]
    and ah,bl
    jz jemp2  //象素点在当前位面的颜色值为0，转到jemp2
    or ch,1  //象素点在当前位面的颜色值为1，ch的相应位置1
    }
jemp2:
asm {
    dec al
    jge loop1
    mov ah,0
    mov al,ch
    }
color.byte=_AX;
return(color);
}
```

5.5　画 扫 描 线

5.5.1　操作步骤

只有写方式0和写方式3适合于画扫描线，而写方式3的速度最快，这里采用了写方式3来实现画扫描线函数。写方式3是VGA特有的，因此这一程序不能移植到EGA上。在写方式3下，

总是用设置重置寄存器的值写入显示存储器，且不需要做设置重置寄存器允许操作。CPU写入的数据经过循环移位后与位屏蔽寄存器的值相与，结果作为位屏蔽码。其操作步骤如下：

①计算扫描线所在的页号(扫描线不会跨页)，及两个端点的地址偏移量，必要时进行换页操作；

②写映象屏蔽寄存器，使所有位面写允许；

③写模式寄存器，选择写方式3；

④将颜色值写入设置重置寄存器；

⑤写循环移位寄存器，不作循环移位；

⑥写位屏蔽寄存器，将其值置为0FFh，以使由CPU写入的数据成为真正的位屏蔽码；

⑦判别两个端点的偏移量是否相同：若相同转到⑧；若不同转到⑨；

⑧根据两端点的位置计算出一个屏蔽码，写入显示存储器，结束；

⑨分最前一个字节、中间若干字节、最后一个字节三段写显示存储器。写最前和最后字节时，需根据端点的位置计算屏蔽码。结束。

5.5.2 程序

系统程序5-4 16.H

```
--------------------------------------------------------------------------------
/*  16色模式下画扫描线  */
void VGA16::scanline(int x1,int x2,int y)
{
unsigned char color=CUR_COLOR.byte;
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下6行，使x1<x2  */
    mov ax,x1
    mov bx,x2
    cmp bx,ax
    jge jemp1
    mov x1,bx
    mov x2,ax
    }
jemp1:
asm {
/*  以下20行，计算扫描线所在的页，及其两个端点的偏移量，分别放在di、si中，同时完成必要的换页操作  */
    mov ax,y
    mul scanleng
    push ax
    mov bx,x1
    shr bx,1
    shr bx,1
    shr bx,1
    add ax,bx
    mov di,ax
```

```
    mov bx,x2
    shr bx,1
    shr bx,1
    shr bx,1
    pop ax
    add ax,bx
    mov si,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
/*  以下6行，写映象屏蔽寄存器，使所有位面写允许  */
    mov dx,SEQUENCE_CTL
    mov al,2
    out dx,al
    inc dx
    mov al,0fh
    out dx,al
 /*  以下6行，置写方式3  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
    mov al,3h
    out dx,al
/*  以下6行，将颜色值写到设置重置寄存器中  */
    dec dx
    mov al,0h
    out dx,al
    inc dx
    mov al,color
    out dx,al
/*  以下6行，将位屏蔽寄存器置为0FFh，以使CPU的写入数据成为真正的位屏蔽码  */
    dec dx
    mov al,08h
    out dx,al
    inc dx
    mov al,0ffh
    out dx,al

    mov ax,G_SEGMENT
    mov es,ax
    cmp si,di  //判断始点和末点是否处于同一字节中
    je inonebyte  //若是转到inonebyte
    }
asm {
```

```
/*  以下6行，写最前的一个字节  */
    mov cx,x1
    and cx,7
    mov al,0ffh
    shr al,cl
    mov ah,es:[di]  //写之前仍需进行一次读操作，以使位屏蔽发生作用
    mov es:[di],al
    mov al,0ffh
    }
middle_byte_loop:
/*  该循环写中间的若干个字节  */
asm {
    inc di
    cmp di,si
    je write_end_byte
    mov es:[di],al  //因为是写全部8位，不需屏蔽，因此写之前无需进行读操作
    jmp middle_byte_loop
    }
write_end_byte:
asm {
/*  以下7行，写最后的一个字节  */
    mov cx,x2
    not cx
    and cx,7
    mov al,0ffh
    shl al,cl
    mov ah,es:[di]
    mov es:[di],al
    jmp end
    }
inonebyte:
/*  以下为始点末点处于同一字节的操作  */
asm {
    mov cx,x1
    and cx,7
    mov al,0ffh
    shr al,cl
    mov cx,x2
    not cx
    and cx,7
    mov ah,0ffh
    shl ah,cl
    and al,ah
    mov ah,es:[di]
    mov es:[di],al
    }
end:
}
--------------------------------------------------------------------------
```

5.6　读 扫 描 线

采用读方式0读扫描线，其寄存器操作步骤与读点完全一样。

系统程序5-5　16.CPP

```
--------------------------------------------------------------------------------
/*  16色模式下读扫描线  */
void VGA16::getscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unsigned byte1;  //始点的地址偏移量
unsigned byte2;  //末点的地址偏移量
unsigned x2;  //末点的横坐标值
char pn=4;  //位面数
asm {
/*  以下4行，计算x2的值  */
    mov ax,x1
    add ax,n
    dec ax
    mov x2,ax
/*  以下20行，计算页号，始点、末点的偏移量，并进行换页  */
    mov ax,y
    mul scanleng
    push ax
    mov bx,x1
    shr bx,1
    shr bx,1
    shr bx,1
    add ax,bx
    mov byte1,ax
    pop ax
    mov bx,x2
    shr bx,1
    shr bx,1
    shr bx,1
    add ax,bx
    mov byte2,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
/*  以下6行，选择读方式0  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
```

```
    mov al,0
    out dx,al
/*  以下4行，将寄存器操作指向读位面选择寄存器  */
    dec dx
    mov al,04h
    out dx,al
    inc dx
/*  以下5行，计算始点所处的位  */
    mov cx,x1
    and cx,7
    mov bl,cl
    mov bh,8
    sub bh,bl

    push ds
    mov ax,G_SEGMENT
    mov ds,ax
    les di,buf
    mov ax,byte2;
    sub ax,byte1;
    add ax,2
    mov es:[di],al
    inc di
    }
loop1:  //位面循环
asm {
    dec pn
    mov al,pn
    out dx,al  //选择读位面
    mov si,byte1
    }
    loop2:  //位面内的字节循环
    asm {
        mov ah,ds:[si]
        inc si
        mov al,ah
        mov cl,bh
        shr ah,cl
        or es:[di],ah
        inc di
        mov cl,bl
        shl al,cl
        mov es:[di],al
        cmp byte2,si
        jge loop2
        inc di
    cmp pn,0
    jne loop1
```

```
    pop ds
    }
}

/*  取16色模式下保存一条扫描线所需缓冲区的大小  */
int VGA16::scanlinesize(int x1,int x2)
{
return ( (x2-x1)/8 + 3 )*4 + 2;
}
```

5.7　写 扫 描 线

采用方式0写扫描线，其寄存器操作步骤与方式0写点基本相同。写扫描线时，要将数据写入所有位面，而不论数据是0还是1，因此不再需要在写之前将原处的数据清零，这是与写点的主要差别。

系统程序5-6　16.CPP

```
/*  16色模式下写扫描线  */
void VGA16::putscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unsigned x2,byte1,byte2;
unsigned char pn=4;
unchar pbn;  //扫描线在每个位面中的字节数
asm {
/*  以下4行，计算x2  */
    mov ax,x1
    add ax,n
    dec ax
    mov x2,ax
/*  以下20行，计算页号、两个端点的偏移地址，并进行换页  */
    mov ax,y
    mul scanleng
    push ax
    mov bx,x1
    shr bx,1
    shr bx,1
    shr bx,1
    add ax,bx
    mov byte1,ax
    mov bx,x2
    shr bx,1
    shr bx,1
    shr bx,1
```

```
    pop ax
    add ax,bx
    mov byte2,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
/*  以下6，行置写模式0  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
    mov al,0h
    out dx,al
/*  以下6行，禁止所有位面的设置重置  */
    dec dx
    mov al,01h
    out dx,al
    inc dx
    mov al,0h
    out dx,al
/*  以下6行，清除循环移位和逻辑运算  */
    dec dx
    mov al,03h
    out dx,al
    inc dx
    mov al,0h
    out dx,al
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf;
    mov al,ds:[si]
    mov pbn,al
/*  以下5行，计算扫描线的在起始地址中的开始位  */
    mov cx,x1
    and cx,7
    mov bl,cl
    mov bh,8
    sub bh,bl
    }
loop1:  //位面循环
asm {
    dec pn
    lds si,buf
    mov al,3
```

```
    sub al,pn
    mul pbn
    add ax,2
    add si,ax
    mov di,byte1
/*  以下8行，写映象屏蔽寄存器，使当前位面写允许，其它位面写禁止  */
    mov dx,SEQUENCE_CTL
    mov al,2
    out dx,al
    inc dx
    mov cl,pn
    mov al,1
    shl al,cl
    out dx,al
 /*  以下4行，将寄存器操作指向位屏蔽寄存器  */
    mov dx,GRAPHICS_CTL
    mov al,08h
    out dx,al
    inc dx
    mov ax,byte1
    cmp ax,byte2
    je inonebyte
/*  以下9行写最前一个字节
    mov al,0ffh
    mov cl,bl
    shr al,cl
    out dx,al
    mov al,ds:[si]
    mov cl,bl
    shr al,cl
    mov ah,es:[di]
    mov es:[di],al
    mov al,0ffh
    out dx,al
    }
    middle_byte_loop:  //该循环写中间若干个字节
    asm {
        inc di
        cmp di,byte2
        je write_end_byte
        mov al,ds:[si]
        mov cl,bh
        shl al,cl
        inc si
        mov ah,ds:[si]
        mov cl,bl
        shr ah,cl
        or al,ah
```

```
        mov es:[di],al
        jmp middle_byte_loop
        }
write_end_byte:
asm {
/*  以下16行，写最后一个字节  */
    mov cx,x2
    not cx
    and cx,7
    mov al,0ffh
    shl al,cl
    out dx,al
    mov al,ds:[si]
    mov cl,bh
    shl al,cl
    inc si
    mov ah,ds:[si]
    mov cl,bl
    shr ah,cl
    or al,ah
    mov ah,es:[di]
    mov es:[di],al
    jmp jemp3
    }
inonebyte:
/*  以下为始点、末点偏移地址相同时的写操作  */
asm {
    mov cx,x1
    and cx,7
    mov al,0ffh
    shr al,cl
    mov cx,x2
    not cx
    and cx,7
    mov ah,0ffh
    shl ah,cl
    and al,ah
    out dx,al
    mov al,ds:[si]
    mov cl,bl
    shr al,cl
    mov ah,es:[di]
    mov es:[di],al
    }
jemp3:
if(pn>0)
    goto loop1;
asm pop ds
}
--------------------------------------------------------------------------
```

5.8 清　　屏

除了置色清屏外，这里还重新定义了在VGABASE类中已经定义了的置零清屏函数。这里采用写方式3实现清屏，其寄存器操作步骤与画扫描线相同。

系统程序5-7　16.CPP

```
--------------------------------------------------------------------------
/*  16色模式下置零清屏  */
void VGA16::cls0()
{
int pn=PAGEN;
display_off();
asm {
/*  写映象屏蔽寄存器，使所有位面写允许  */
    mov dx,SEQUENCE_CTL
    mov al,2
    out dx,al
    inc dx
    mov al,0fh
    out dx,al
/*  置写方式3  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
    mov al,3h
    out dx,al
/*  清循环移位寄存器  */
    dec dx
    mov al,3h
    out dx,al
    mov al,0h
    inc dx
    out dx,al
/*  将位屏蔽寄存器置为0FFh  */
    dec dx
    mov al,08h
    out dx,al
    mov al,0ffh
    inc dx
    out dx,al
/*  将0写入设置重置寄存器  */
    dec dx
    mov al,0h
    out dx,al
    inc dx
    mov al,0
    out dx,al
    mov si,0
```

```
    }
loop:
    Select_Page(_SI);
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0
    mov ax,0FFFFh
    mov cx,8000h
    cld
    rep stosw
    inc si
    cmp si,pn
    jne loop
    }
display_on();
}

/*  16色模式下置色清屏  */
void VGA16::cls()
{
int pn=PAGEN;
unsigned char color=CUR_COLOR.byte;
asm {
/*  写映象屏蔽寄存器，使所有位面写允许  */
    mov dx,SEQUENCE_CTL
    mov al,2
    out dx,al
    inc dx
    mov al,0fh
    out dx,al
/*  置写方式3  */
    mov dx,GRAPHICS_CTL
    mov al,05h
    out dx,al
    inc dx
    mov al,3h
    out dx,al
/*  清循环移位寄存器  */
    dec dx
    mov al,3h
    out dx,al
    mov al,0h
    inc dx
    out dx,al
/*  将位屏蔽寄存器置为0FFh  */
    dec dx
    mov al,08h
```

```
    out dx,al
    mov al,0ffh
    inc dx
    out dx,al
/*  将颜色值写入设置重置寄存器  */
    dec dx
    mov al,0h
    out dx,al
    inc dx
    mov al,color
    out dx,al
    mov si,0
    }
loop:
    Select_Page(_SI);
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0
    mov ax,0ffffh
    mov cx,8000h
    cld
    rep stosw
    inc si
    cmp si,pn
    jne loop
    }
}
```

5.9　调色板操作

5.9.1　程序

16色模式下存在两个色彩查找表，第一个色彩查找表由属性控制器提供，它用于将4位16态的颜色索引值转换位8位256态的颜色索引值，然后输出到第二个色彩查找表即DAC色彩查找表，最终由DAC色彩查找表确定出16色模式下每个颜色索引值所对应的真实颜色值。属性控制器以两种方式维持这第一个色彩查找表，这里只实现了其中一种方式，即调色板寄存器的操作。调色板寄存器的读和写可用两种方式来实现，一是直接操作寄存器，二是调用BIOS功能，直接操作寄存器需要在垂直回扫期间进行，编程比较麻烦，因此这里采用了第二种实现方式，即调用BIOS功能。

系统程序5-8　16.CPP

```
/*************************************************************************
功能：置调色板寄存器
输入参数：idx16＝4位的颜色索引值
```

```
          idx64＝8位的颜色索引值，其高两位无效
返回值：无
*************************************************************************/
void VGA16::setpalette(unchar idx16,unchar idx64)
{
asm {
    mov bl,idx16
    and bl,0fh
    mov bh,idx256
    and bh,3fh
    mov ax,1000h
    int 10h
    }
}

/*************************************************************************
功能：读调色板寄存器
输入参数：idx16＝4位的颜色索引值
返回值：unchar＝8位的颜色索引值，其高两位总为零
*************************************************************************/
unchar VGA16::getpalette(unchar idx16)
{
asm {
    mov bl,idx16
    and bl,0fh
    mov ax,1007h
    int 10h
    }
return _BH;
}
```

5.9.2 使用方式

在16色模式下，调色工作最终还是要通过DAC色彩查找表来完成，但在操作DAC时首先需要获得4位颜色索引值与8位颜色索引值之间的对应关系，而不能直接根据4位的颜色索引值来操作DAC，有三种方式来获得这一对应关系，下面将通过三段示例性的程序来说明这三种方式，这三段程序都完成相同的工作：将16色模式下的16个颜色索引值设置为16级灰度值。

1. 采用调色板寄存器的缺省状态

在屏幕初始化之后，调色板寄存器会装入一组缺省值，在大多数VGA上这组缺省值是相同的，见表5-1。

表5-1 调色板寄存器缺省值

4位索引值	8位索引值	真实颜色
0	0	黑
1	1	蓝
2	2	绿
3	3	青蓝
4	4	红
5	5	洋红
6	14h	棕
7	7	白
8	38h	深灰
9	39h	亮蓝
0Ah	3Ah	亮绿
0Bh	3Bh	亮清蓝
0Ch	3Ch	亮红
0Dh	3Dh	亮洋红
0Eh	3Eh	黄
0Fh	3Fh	亮白

示例程序5-1

```
void main()
{
640_480_16 A;
A.init();
unchar idx[]={0,1,2,3,4,5,0x14,7,0x38,0x39,0x3a,0x3b,0x3c,0x3d,0x3e,0x3f};
for(int i=0;i<16;i++)
    A.setdac(idx[i],i*16,i*16,i*16);
}
```

2. 读调色板寄存器

示例程序5-2

```
void main()
{
640_480_16 A;
A.init();
for(int i=0;i<16;i++)
    A.setdac(A.getpalette(i),i*16,i*16,i*16);
}
```

3. 写调色板寄存器

示例程序5-3

```
void main()
{
640_480_16 A;
A.init();
for(int i=0;i<16;i++)
    {
    A.setpalette(i,i);
    A.setdac(i,i*16,i*16,i*16);
```

```
    }
}
```

5.10 参 数 设 置

三个16色图形模式类，各定义一个构造函数用于参数设置。

系统程序5-9 16.CPP

```
--------------------------------------------------------------------------
/*  640×480×16色参数设置  */
_640_480_16::_640_480_16()
{
SCANLENG=80;
  //每条扫描线在一个位面中所占字节数，在16色模式下，其等于水平象素数除以8
WIDE=640;  //水平象素数
HIGH=480;  //垂直象素数
PAGEN=1;  //所占用的存储器页数，16色模式下每页256k
VESAmodeNo=0x0012;  //标准VGA模式号
}

/*  800×600×16色参数设置  */
_800_600_16::_800_600_16()
{
SCANLENG=100;
WIDE=800;
HIGH=600;
PAGEN=1;
VESAmodeNo=0x0102;  //VESA模式号
}

/*  1024×768×16色参数设置  */
_1024_768_16::_1024_768_16()
{
SCANLENG=128;
WIDE=1024;
HIGH=768;
PAGEN=2;
VESAmodeNo=0x0104;
}
--------------------------------------------------------------------------
```

第 6 章　真彩色模式的图形操作

6.1　概　　述

真彩色模式能够表现极为丰富、细腻的色彩，其色彩表现能力已基本达到了 CRT 显示器的极限，而且在该模式下可以直接对每个象素点的三基色值进行操作，而不需要通过色彩查找表来间接确定象素颜色，也不需要考虑全屏颜色数的限制，与 16 色、256 色之类的间接色彩模式相比，真彩色模式的图形显示性能有了一个本质的提高。真彩色模式在计算机图形显示系统中是一种较新的模式，它在目前还没有得到广泛的应用，随着硬件价格的降低、速度的提高，可以预见真彩色及高彩色模式将成为未来计算机图形显示的主流，16 色和 256 色模式将和早期的单色、4 色模式一样会被逐渐淘汰。

从色彩表现能力及色彩处理方式来看，24位真彩色模式有可能是VGA的终极模式，但其3字节的颜色数据长度是一个明显的缺点，3个字节即给显示存储器的组织带来了麻烦，也不便于程序处理，因此在VGA上有可能出现32位的真彩色模式。

程序实现了对两种24位真彩色模式的支持：320×200×16M色和640×480×16M色。这两种模式的存储器分页情况都为行外分页，这里将按行外分页的情况进行编程。事实上，在3字节的真彩色模式中不可能出现不分页或行内分页的情况，因为行内分页必然就会导致点内分页，因此所实现的程序能够用于更高分辨率的24位真彩色模式。

24位真彩色模式下的显示存储器结构参见2.1节中的有关内容。在该模式下访问显示存储器不需要进行寄存器操作。

24位真彩色模式非常适合于三维实体造型、三维动画方面的应用，因此在真彩色模式类中增加了一些与逼真物体的明暗处理有关的颜色变换函数。

6.2　编 程 方 案

与24位真彩色模式相关的类有3个：VGA16M、_320_200_16M、_640_480_16M，这三个类的继承关系见图3-1，下面给出它们的说明。

系统程序6-1　VGA16M.H

```
--------------------------------------------------------------------------
/*  16M色模式类  */
class VGA16M : public VGABASE
    {
public:
    void putpixel(int x,int y);
    union COLOR getpixel(int x,int y);
    void cls(void);
    int scanlinesize(int x1,int x2);
    void scanline(int x1,int x2,int y);
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);
```

```
    void putpixel(int x,int y,COLOR color)
        { CUR_COLOR=color; putpixel(x,y); }
    void scanline(int x1,int x2,int y,COLOR color)
        { CUR_COLOR=color; scanline(x1,x2,y); }
    void cls(COLOR color)
        { CUR_COLOR=color; cls(); }
/*  以下为真彩色模式下新增的15个函数  */
    void en_putpixel(int x,int y,unchar enl,unchar redu);
    void en_scanline(int x1,int x2,int y,unchar enl,unchar redu);
    void en_bar(int x1,int y1,int x2,int y2,unchar enl,unchar redu);

    void re_putpixel(int x,int y);
    void re_scanline(int x1,int x2,int y);
    void re_bar(int x1,int y1,int x2,int y2);
    void de_putpixel(int x,int y);
    void de_scanline(int x1,int x2,int y);
    void de_bar(int x1,int y1,int x2,int y2);

    void re_putpixel(int x,int y,COLOR col)
        { CUR_COLOR=col; re_putpixel(x,y); }
    void re_scanline(int x1,int x2,int y,COLOR col)
        { CUR_COLOR=col; re_scanline(x1,x2,y); }
    void re_bar(int x1,int y1,int x2,int y2,COLOR col)
        { CUR_COLOR=col; re_bar(x1,y1,x2,y2); }
    void de_putpixel(int x,int y,COLOR col)
        { CUR_COLOR=col; de_putpixel(x,y); }
    void de_scanline(int x1,int x2,int y,COLOR col)
        { CUR_COLOR=col; de_scanline(x1,x2,y); }
    void de_bar(int x1,int y1,int x2,int y2,COLOR col)
        { CUR_COLOR=col; de_bar(x1,y1,x2,y2); }
    };

/*  320×200×16M色模式类  */
class _320_200_16M : public VGA16M
    {
public:
    _320_200_16M();
    };

/*  640×480×16M色模式类  */
class _640_480_16M : public VGA16M
    {
public:
    _640_480_16M();
    };
--------------------------------------------------------------------------------
```

6.3　点　操　作

6.3.1　地址计算

Page=(ScanLeng*Y+X*3)/10000H
Offset=(ScanLeng*Y+X*3)%10000H

Page——象素点所在的页号

Offset——象素点在页中偏移量

ScanLeng——条扫描线所占字节数

X、Y——象素点的坐标位置

在其它色彩模式中，扫描线所占字节数与屏幕水平象素数之间有明确的对应关系，因而可以直接用水平象素数来进行地址计算，但在真彩色模式下，由于要避免点内分页，使得扫描线所占字节数与水平象素数之间没有确定的关系，因而只能用扫描线所占字节数来计算地址。

在320×200×16M色模式下，扫描线所需的字节数为960，但占用了1024；在640×480×16M色模式下，扫描线所需的字节数为1920，占用了2048。由这两种模式的情况似乎可以找到一个规律，即首先算出扫描线所需的字节数，然后向上取整到能整除64k的某个数上，有可能采用的能整除64k的数只有如下几个：1024、2048、4096、8192。上面所给出的两种模式确实符合这一规律，但当扩展到800×600×16M色模式时，这一规律就得不到正确的结果，该模式下每条扫描线所需字节数为2400，整屏所需容量约为1.4M，按上述规律进行处理后，每条扫描线就要占用4096个字节，整屏所需的容量就要达到2.4M，这需要4M的显示卡来支持，而实际上一些2M的显示卡都提供了对这一模式的支持，实现这种支持的方式为，将每条扫描线所占的字节数定为2850，而非4096，这时尽管分页会出现在扫描线所占用的字节中，但不会出现在扫描线实际使用的字节中，因而也能避免点内分页。所以，各种16M色模式下的ScanLeng参数只能直接获取，而不能通过计算得到。

6.3.2　程序

系统程序6-2　16M.CPP

```
--------------------------------------------------------------------------
/*  24位色模式下写一个象素点  */
void VGA16M::putpixel(int x,int y)
{
int scanleng=SCANLENG;  //扫描线所占字节数
int cur_page=CUR_PAGE;  //当前页
unchar *rgb=CUR_COLOR.rgb;  //象素点的颜色值，共3个字节
asm {
/*  以下10行，计算页号(dx)和偏移量(di)，并在必要时进行换页  */
    mov ax,y
    mul scanleng
    add ax,x
    shl x,1
    add ax,x
    mov di,ax
```

```
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
jemp1:
asm {
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    mov cx,3
    lds si,rgb
    cld
    rep movsb  //用串传送指令写入象素点的3个字节
    pop ds
    }
}

/*  24位色模式下读一个象素点  */
COLOR VGA16M::getpixel(int x,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
COLOR color={0};
unsigned char *rgb=color.rgb;

asm {
/*  以下10行，进行地址计算和换页  */
    mov ax,y
    mul scanleng
    add ax,x
    shl x,1
    add ax,x
    mov si,ax
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
jemp1:
asm {
    push ds
    mov ax,G_SEGMENT
    mov ds,ax
    mov cx,3
    les di,rgb
    cld
    rep movsb  //用串传送指令读出象素点的3个字节
    pop ds
    }
```

```
return(color);
}
-------------------------------------------------------------------------
```

6.4　扫描线操作

真彩色模式中，每个象素点的颜色用3个字节表示，机器中的串传送指令无法一次处理3个字节，因而不能用带重复前缀的存串指令一次画出一条扫描线，这影响了该模式下画扫描线的速度。同样也不能用带重复前缀的存串指令来完成置色清屏操作，这时直接实现置色清屏操作在速度上就不能获得太大的好处，因此该模式下的置色清屏函数通过调用画扫描线函数来实现。读和写扫描线操作仍旧针对单个字节进行处理，因而它们的速度不受影响。

系统程序6-3　16M.CPP

```
-------------------------------------------------------------------------
/*  24位色模式下画扫描线  */
void VGA16M::scanline(int x1,int x2,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unsigned char *rgb=CUR_COLOR.rgb;
asm {
/*  以下12行，使x1<x2，并计算出扫描线长度，入栈  */
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
    mov x1,bx
    }
jemp1:
asm {
    inc ax
    push ax
/*  10行计算扫描线始点的地址，并进行换页*/
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    pop bx  //扫描线长度出栈，存于bx
```

```
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    cld
    }
loop1:  //该循环中一次画一个象素点
asm {
    lds si,rgb
    mov cx,3
    rep movsb  //串传送指令，一次画一个象素点
    dec bx
    jnz loop1
    pop ds
    }
}

/*  24位色模式下置色清屏  */
void VGA16M::cls()
{
int i,x2=WIDE-1;
for(i=0;i<HIGH;i++)
    scanline(0,x2,i);
}

/*  24位色模式下读扫描线  */
void VGA16M::getscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下10行，计算扫描线始点地址，并进行换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov si,ax
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
jemp1:
asm {
    push ds
/*  以下3行，将扫描线长度乘3，得到扫描线的字节数，存于cx  */
    mov cx,n
    shl cx,1
    add cx,n
```

```
    les di,buf
    mov ax,G_SEGMENT
    mov ds,ax
    cld
    rep movsb  //带重复前缀的串传送指令，一次读出整个扫描线
    pop ds
    }
}

/*  24位色模式下写扫描线  */
void VGA16M::putscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下10行，计算扫描线始点地址，并进行换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov di,ax
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
jemp1:
asm {
    push ds
    mov cx,n
    shl cx,1
    add cx,n
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf
    cld
    rep movsb  //一次写入整个扫描线
    pop ds
    }
}

/*  取24位色模式下保存一条扫描线所需的缓冲区大小  */
int VGA16M::scanlinesize(int x1,int x2)
{
int size;
size=(x2-x1+1)*3;
if( size&0x0001 )
    size++;
```

```
return(size);
}
-------------------------------------------------------------------------
```

6.5 颜 色 变 换

这里实现了一些对象素点的已有颜色进行变换的函数，这些函数可用于对逼真物体进行明暗处理。这些函数分为三组：亮度变换、叠加写入、去除叠加。这三组函数都分别针对单个象素点、扫描线、矩形块进行操作。

所有颜色变换都针对象素点的三个基色分别进行。在24位色模式下，每个基色的取值范围为0～255，如果变换的结果超出该范围，就将结果置为最大值或最小值。

6.5.1 亮度变换

这组函数用于将屏幕上某些象素点的颜色加亮或减暗，变换中保持象素点三基色的比例不变。

进行变换时需给定一个变换系数，该系数应为一个实数，变换系数乘以象素点的原颜色值即得到变换的结果颜色值。为避免浮点运算，加快速度，这里用两个整型数来代替变换系数，一个为增强系数，一个为减弱系数，原颜色值乘以增强系数除以减弱系数形成变换的结果颜色值，例如，增强系数＝5，减弱系数＝4，则变换后的亮度为原来的1.25倍；增强系数＝3，减弱系数＝4，则变换后的亮度为原来的0.75。

系统程序6-4 16M.CPP

```
-------------------------------------------------------------------------
/*************************************************************************
功能：变换一个象素点的亮度
输入参数： x＝象素点的横坐标
           y＝象素点的纵坐标
           enl＝亮度增强系数
           redu＝亮度减弱系数
返回值：无
*************************************************************************/
void VGA16M::en_putpixel(int x,int y,unchar enl,unchar redu)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下10行，计算地址，换页*/
    mov ax,y
    mul scanleng
    add ax,x
    shl x,1
    add ax,x
    mov di,ax
    cmp dx,cur_page
    je jemp1
    }
```

```
    Select_Page(_DX);
jemp1:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov cx,3h
    }
loop1:  //针对颜色值的每个字节的循环
asm {
    mov al,es:[di]  //读出象素点原颜色中的某个基色值，放在al中
/*  以下6行，对读出的基色值进行变换，结果放在bl中  */
    mul enl  //首先将原颜色放大，进位放在ah中
    mov bl,0ffh
    cmp redu,ah  //减弱系数是否大于ah
    jc jemp2  //是：执行以下2行；否：bl=0FFh，即在该基色上达到最亮
    div redu  //放大后的值除以减弱系数
    mov bl,al
    }
jemp2:
asm {
    mov es:[di],bl  //将变换的结果写回象素点
    inc di
    dec cx
    jnz loop1
    }
}

/************************************************************************
功能：变换一条扫描线的亮度
输入参数：  x1＝扫描线始点的横坐标
            x2＝扫描线末点的横坐标
            y＝扫描线的纵坐标
            enl＝亮度增强系数
            redu＝亮度减弱系数
返回值：无
************************************************************************/
void VGA16M::en_scanline(int x1,int x2,int y,unchar enl,unchar redu)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
asm {
/*  以下15行，使x1<x2，计算扫描线的字节数，入栈  */
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
```

```
    mov x1,bx
    }
jemp1:
asm {
    inc ax
    mov bx,ax
    shl ax,1
    add ax,bx
    push ax
/*  以下10行，计算地址、换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    pop cx  //扫描线字节数出栈，放于cx
    }
loop1:  //针对扫描线上每个字节的循环
asm {
    mov al,es:[di]  //读出扫描线上某个点的某个基色
/*  以下6行，对读出的基色进行变换，结果放在bl中 */
    mul enl
    mov bl,0ffh
    cmp redu,ah
    jc jemp3
    div redu
    mov bl,al
    }
jemp3:
asm {
    mov es:[di],bl
    inc di
    dec cx
    jnz loop1
    }
}
```

```
/*************************************************************************
功能：增强或减弱一个矩形区域的亮度
输入参数：  x1＝矩形左上端点的横坐标
            y1＝矩形左上端点的纵坐标
            x2＝矩形右下端点的横坐标
            y2＝矩形右下端点的纵坐标
            enl＝亮度增强系数
            redu＝亮度减弱系数
返回值：无
*************************************************************************/
void VGA16M::en_bar(int x1,int y1,int x2,int y2,unchar enl,unchar redu)
{
int i;
for(i=y1;i<=y2;i++)
    en_scanline(x1,x2,i,enl,redu);
}
```

6.5.2　叠加写入

这组函数用于将某一颜色值加到某个或某些象素点上。叠加后象素点的亮度将增强，三基色的比例也将发生变化。

这组函数有两个版本，一个为直接传递颜色值，一个为间接传递颜色值，这里实现的是后一个版本，前者已在VGA16M类说明中以内部函数的形式实现。

系统程序6-5　16M.CPP

```
-------------------------------------------------------------------------
/*************************************************************************
功能：将当前颜色叠加写到一个象素点上
输入参数：  x＝象素点的横坐标
            y＝象素点的纵坐标
返回值：无
*************************************************************************/
void VGA16M::re_putpixel(int x,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unchar *rgb=CUR_COLOR.rgb;  //取当前颜色用于叠加
asm {
/*  以下10行，计算地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x
    shl x,1
    add ax,x
    mov di,ax
    cmp dx,cur_page
    je jemp1
```

```
    }
    Select_Page(_DX);
jemp1:
asm {
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    lds si,rgb
    mov cx,3h
    }
loop1:  //针对象素点每个基色的循环
asm {
    mov al,es:[di]  //读出象素点的某个原基色，放在al中
    add al,ds:[si]  //加上叠加色中相应的基色
    jnc jemp2  //是否有进位，即变换结果是否向上超出范围
    mov al,0ffh  //是：将变换结果取为最大值0FFh
    }
jemp2:
asm {
    mov es:[di],al  //将变换结果写回象素点
    inc di
    inc si
    dec cx
    jnz loop1
    pop ds
    }
}

/*************************************************************************
功能：将当前颜色值叠加写到一条扫描线的每个象素点上
输入参数：  x1＝扫描线始点的横坐标
            x2＝扫描线末点的横坐标
            y ＝扫描线的纵坐标
返回值：无
*************************************************************************/
void VGA16M::re_scanline(int x1,int x2,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unchar *rgb=CUR_COLOR.rgb;
asm {
/* 以下12行，使x1<x2；计算扫描线长度，入栈  */
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
```

```
    mov x1,bx
    }
jemp1:
asm {
    inc ax
    push ax
/*  以下10行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    pop bx  //扫描线长度出栈，放于bx
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    lds si,rgb
    mov cx,3h
    }
loop1:
asm {
/*  以下8行，完成某个点的某个基色的读出、变换和写回  */
    mov al,es:[di]
    add al,ds:[si]
    jnc jemp3
    mov al,0ffh
    }
jemp3:
asm {
    mov es:[di],al
    inc di
    inc si
    dec cx
    jnz loop1  //一个点的各个基色的循环
    mov cx,3h
    sub si,3h
    dec bx
    jnz loop1  //各个象素点的循环
    pop ds
   }
}
```

```
/*************************************************************************
功能：将当前颜色叠加写到一个矩形区域中的每个象素点上
输入参数：  x1＝矩形左上端点的横坐标
            y1＝矩形左上端点的纵坐标
            x2＝矩形右下端点的横坐标
            y2＝矩形右下端点的纵坐标
返回值：无
*************************************************************************/
    void VGA16M::re_bar(int x1,int y1,int x2,int y2)
    {
    int i;
    for(i=y1;i<=y2;i++)
        re_scanline(x1,x2,i);
    }
```

6.5.3 去除叠加

这组函数用于从某个或某些象素点的颜色值中减去某一颜色值，即去除叠加在象素点上的某一颜色。去叠加后象素点的亮度将减弱，三基色的比例也将发生变化。

这组函数有两个版本，一个为直接传递颜色值，一个为间接传递颜色值，这里实现的是后一个版本，前者已在VGA16M类说明中以内部函数的形式实现。

系统程序6-6 16M.CPP

```
-------------------------------------------------------------------------
/*************************************************************************
功能：去除叠加在单个象素点上的某一颜色值
输入参数：  x＝象素点的横坐标
            y＝象素点的纵坐标
返回值：无
*************************************************************************/
void VGA16M::de_putpixel(int x,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unchar *rgb=CUR_COLOR.rgb;  //以当前颜色为去叠加色
asm {
/*  以下10行，计算地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x
    shl x,1
    add ax,x
    mov di,ax
    cmp dx,cur_page
    je jemp1
    }
    Select_Page(_DX);
```

```
jemp1:
asm {
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    lds si,rgb
    mov cx,3h
    }
loop1:  //象素点每个基色的循环
asm {
    mov al,es:[di]  //读出象素点的某个基色
    sub al,ds:[si]  //将所读出的基色减去去叠加色中相应的基色
    jnc jemp2  //是否有借位，即变换结果是否向下超出范围
    mov al,0h  //是：将变换结果取为最小值0
    }
jemp2:
asm {
    mov es:[di],al  //将变换结果写回象素点
    inc di
    inc si
    dec cx
    jnz loop1
    pop ds
    }
}

/*************************************************************************
功能：去除叠加在扫描线上的某一颜色值
输入参数：  x1＝扫描线始点的横坐标
            x2＝扫描线末点的横坐标
            y ＝扫描线的纵坐标
返回值：无
*************************************************************************/
void VGA16M::de_scanline(int x1,int x2,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
unchar *rgb=CUR_COLOR.rgb;
asm {
/*  以下12行，使x1<x2；计算扫描线的长度，入栈  */
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
    mov x1,bx
    }
```

```
jemp1:
asm {
    inc ax
    push ax
/*  以下10行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    shl x1,1
    add ax,x1
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    pop bx  //扫描线长度出栈
    push ds
    mov ax,G_SEGMENT
    mov es,ax
    lds si,rgb
    mov cx,3h
    }
loop1:
asm {
/*  以下8行，完成某个点的某个基色的读出、变换和写回  */
    mov al,es:[di]
    sub al,ds:[si]
    jnc jemp3
    mov al,0h
    }
jemp3:
asm {
    mov es:[di],al
    inc di
    inc si
    dec cx
    jnz loop1  //针对一个点的各个基色的循环
    mov cx,3h
    sub si,3h
    dec bx
    jnz loop1  //针对各个象素点的循环
    pop ds
    }
```

```
}
/*************************************************************************
功能：去除叠加在一个矩形区域上的某一颜色值
输入参数：  x1＝矩形左上端点的横坐标
            y1＝矩形左上端点的纵坐标
            x2＝矩形右下端点的横坐标
            y2＝矩形右下端点的纵坐标
返回值：无
*************************************************************************/
void VGA16M::de_bar(int x1,int y1,int x2,int y2)
{
int i;
for(i=y1;i<=y2;i++)
    de_scanline(x1,x2,i);
}
```

6.6　参 数 设 置

系统程序6-6　16M.CPP

```
/*  320×200×16M色参数设置  */
_320_200_16M::_320_200_16M()
{
SCANLENG=1024;  //每条扫描线所占字节数
WIDE=320;  //屏幕水平象素数
HIGH=200;  //屏幕垂直象素数
PAGEN=4;  //所需的显示存储器页数
VESAmodeNo=0x010f;  //VESA模式号
}

/*  640×480×16M色参数设置  */
_640_480_16M::_640_480_16M()
{
SCANLENG=2048;
WIDE=640;
HIGH=480;
PAGEN=15;
VESAmodeNo=0x0112;
}
```

第 7 章　高彩色模式的图形操作

7.1　概　　述

高彩色模式也是一种直接色彩模式，虽然它所能表现的色彩不如真彩色模式丰富，但它已经达到了非常逼真地再现各种真实图象的水平，它所需的存储器容量比真彩色模式要小，因而它能具有较高的分辨率、较快的速度或较低的价格，而且它的双字节颜色值也更便于处理，因此在很多情况下高彩色模式比真彩色模式往往具有更大的吸引力。在目前已有一些多媒体、三维动画、CAD等方面的应用都采用了高彩色模式。

高彩色模式分为两类：15位高彩色模式和16位高彩色模式，本书实现了对这两类共8种高彩色图形模式的支持，它们是：320×200×32K色、512×480×32K色、640×480×32K色、800×600×32K色，320×200×64K色、512×480×64K色、640×480×64K色、800×600×64K色。

这8种模式包含了两种显示存储器分页情况：行外分页和行内分页，在这种模式下不可能出现不分页的情况，这里将按照行内分页的情况进行编程。

高彩色模式下的显示存储器结构见2.1节中的有关内容。在这种模式下访问显示存储器不需要进行寄存器操作。但在这种模式下需要将三个基色值合成到2个字节中，然后才能写入显示存储器，与最简单的256色模式相比，这是该模式下编程唯一较麻烦的地方。VGABASE类中定义的颜色转换函数和当前颜色设置函数不能用于高彩色模式，这里需重新定义这些函数。

7.2　编 程 方 案

15位色模式和16位色模式在显示存储器的结构上只有一个差别，就是2字节的颜色值中各位的定义不同，由于采用了COLOR联合及间接颜色传递方式，这一差别只被局限在颜色转换函数及颜色设置函数中，而使得其它的图形操作函数在这两类模式中不存在差别。

这里定义了如下11个与高彩色模式相关的类：VGAHIGH、VGA32K、VGA64K、_320_200_32K、_512_480_32K、_640_480_32K、_800_600_32K、_320_200_64K、_512_480_64K、_640_480_64K、_800_600_64K。这些类的继承关系见图3-1。下面给出这些类的说明。

系统程序7-1　VGAHIGH.H

```
--------------------------------------------------------------------------
/*  高彩色模式类  */
class VGAHIGH : public VGABASE
    {
public:
    void putpixel(int x,int y);
    union COLOR getpixel(int x,int y);
    void cls(void);
    int scanlinesize(int x1,int x2);
    void scanline(int x1,int x2,int y);
```

```
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);

    void putpixel(int x,int y,COLOR color)
        { CUR_COLOR=color; putpixel(x,y); }
    void scanline(int x1,int x2,int y,COLOR color)
        { CUR_COLOR=color; scanline(x1,x2,y); }
    void cls(COLOR color)
        { CUR_COLOR=color; cls(); }
    } ;

/*  32K色模式类  */
class VGA32K : public VGAHIGH
    {
public:
/*  该类重定义了如下4个颜色转换和颜色设置函数  */
    void setcolor(unchar r,unchar g,unchar b);
    void setcolor(unchar light)
        { setcolor(light,light,light); }
    COLOR setcolorto(unchar r,unchar g,unchar b);
    COLOR setcolorto(unchar light)
        { return setcolorto(light,light,light); }
    } ;

/*  64K色模式类  */
class VGA64K : public VGAHIGH
    {
public:
/*  该类重定义了如下4个颜色转换和颜色设置函数  */
    void setcolor(unchar r,unchar g,unchar b);
    void setcolor(unchar light)
        { setcolor(light,light,light); }
    COLOR setcolorto(unchar r,unchar g,unchar b);
    COLOR setcolorto(unchar light)
        { return setcolorto(light,light,light); }
    } ;

/*  320×200×32K图形模式类  */
class _320_200_32K : public VGA32K
    {
public:
    _320_200_32K();
    };

/*  320×200×64K图形模式类  */
class _320_200_64K : public VGA64K
    {
public:
```

```
    _320_200_64K();
    };

/*  512×480×32K图形模式类  */
class _512_480_32K : public VGA32K
    {
public:
    _512_480_32K();
    };

/*  512×480×64K图形模式类  */
class _512_480_64K : public VGA64K
    {
public:
    _512_480_64K();
    };

/*  640×480×32K图形模式类  */
class _640_480_32K : public VGA32K
    {
public:
    _640_480_32K();
    };

/*  640×480×64K图形模式类  */
class _640_480_64K : public VGA64K
    {
public:
    _640_480_64K();
    };

/*  800×600×32K图形模式类  */
class _800_600_32K : public VGA32K
    {
public:
    _800_600_32K();
    };

/*  800×600×64K图形模式类  */
class _800_600_64K : public VGA64K
    {
public:
    _800_600_64K();
    };
--------------------------------------------------------------------------------
```

7.3　颜色转换及当前颜色设置

颜色转换函数用于将高彩色模式下的原始颜色值转换为COLOR联合中的2个字节，当前颜色设置函数首先进行颜色转换然后将结果赋给CUR_COLOR，COLOR联合中的2个字节的数据格式与显示存储器中每个象素的数据格式一致。高彩色模式下的原始颜色值为3个字节，每个字节表示一种基色的亮度，为方便处理并保持与真彩色模式的统一，每个基色的取值范围定为0～255，因此在颜色转换中还需将8位的基色亮度值转换为5位或6位。15位色和16位色的颜色转换方式是不同的，需要分别进行定义。

7.3.1　15位色模式

15位颜色数据的格式为：位10～位14，定义蓝色的亮度；位5～位9，定义绿色的亮度；位0～位4，定义红色的亮度。每种基色亮度值的取值范围为：0～31，31为最亮。

系统程序7-2　HIGH.CPP

```
--------------------------------------------------------------------------
/************************************************************************
功能：32K色颜色转换
输入参数：  red＝原始颜色中红色的亮度(0～255)
            green＝原始颜色中绿色的亮度(0～255)
            blue＝原始颜色中蓝色的亮度(0～255)
返回值：COLOR＝转换后的颜色值
************************************************************************/
COLOR VGA32K::setcolorto(unchar red,unchar green,unchar blue)
{
COLOR color={0L};
asm {
    mov bx,0  //bx用于存放转换结果
/*  将原色中的蓝色值除以8，然后放到bx的低5位  */
    mov al,blue
    mov cl,3
    shr ax,cl
    and ax,0000000000011111b
    or bx,ax
/*  将原色中的绿色值除以8，放到bx的位5～位9  */
    mov al,green
    mov cl,2
    shl ax,cl
    and ax,0000001111100000b
    or bx,ax
/*  将原色中的红色值除以8，放到bx的位10～位14  */
    mov al,red
    mov cl,7
    shl ax,cl
    and ax,0111110000000000b
    or bx,ax
    }
color.word=_BX;
```

```
return(color);
}

/*************************************************************************
功能：32K色当前颜色设置
输入参数：  red＝原始颜色中红色的亮度(0～255)
            green＝原始颜色中绿色的亮度(0～255)
            blue＝原始颜色中蓝色的亮度(0～255)
返回值：无
    转换结果赋给保存当前颜色值的变量CUR_COLOR。
*************************************************************************/
void VGA32K::setcolor(unchar red,unchar green,unchar blue)
{
unsigned col;
asm {
    mov bx,0  //bx用于存放转换结果
/*  转换蓝色  */
    mov al,blue
    mov cl,3
    shr ax,cl
    and ax,0000000000011111b
    or bx,ax
/*  转换绿色  */
    mov al,green
    mov cl,2
    shl ax,cl
    and ax,0000001111100000b
    or bx,ax
/*  转换红色  */
    mov al,red
    mov cl,7
    shl ax,cl
    and ax,0111110000000000b
    or bx,ax
    }
col=_BX;  //CUR_COLOR是类的成员，无法正确地将寄存器变量的值直接赋给它
CUR_COLOR.word=col;
}
-------------------------------------------------------------------------
```

7.3.2 16位色模式

16位颜色数据的格式为：位11～位15，共5位，定义蓝色的亮度；位5～位10，共6位，定义绿色的亮度；位0～位4，共5位，定义红色的亮度。蓝、红两基色亮度值的取值范围为：0～31，31为最亮。绿色亮度值的取值范围为：0～63，63为最亮。

系统程序7-3　HIGH.CPP

```
--------------------------------------------------------------------------
/*  64K色模式下颜色转换  */
COLOR VGA64K::setcolorto(unchar red,unchar green,unchar blue)
{
COLOR color={0L};
asm {
    mov bx,0  //bx存放转换结果
/*  蓝色值除以8，放到bx的低5位  */
    mov al,blue
    mov cl,3
    shr ax,cl
    and ax,0000000000011111b
    or bx,ax
/*  绿色值除以4，放到bx的中间6位  */
    mov al,green
    mov cl,3
    shl ax,cl
    and ax,0000011111100000b
    or bx,ax
/*  红色值除以8，放到bx的高5位  */
    mov al,red
    mov cl,8
    shl ax,cl
    and ax,1111100000000000b
    or bx,ax
    }
color.word=_BX;
return(color);
}

/*  64K色模式下当前颜色设置  */
void VGA64K::setcolor(unchar red,unchar green,unchar blue)
{
unsigned col;
asm {
    mov bx,0
/*  转换蓝色  */
    mov al,blue
    mov cl,3
    shr ax,cl
    and ax,0000000000011111b
    or bx,ax
/*  转换绿色  */
    mov al,green
    mov cl,3
    shl ax,cl
    and ax,0000011111100000b
```

```
    or bx,ax
/*  转换红色  */
    mov al,red
    mov cl,8
    shl ax,cl
    and ax,1111100000000000b
    or bx,ax
    }
col=_BX;
CUR_COLOR.word=col;
}
```

7.4 点 操 作

7.4.1 地址计算

Page＝(Width*Y*2＋X*2)/10000H

Offset＝(Width*Y*2＋X*2)%10000H

Page——象素点所在的页号

Offset——象素点在页中的偏移量

Width——水平象素数

X、Y——象素点的坐标位置

在高彩色模式下，屏幕水平象素数乘2即等于一条扫描线所占字节数。

7.4.2 程序

系统程序7-4 HIGH.CPP

```
/*  高彩色模式下写单个象素点  */
void VGAHIGH::putpixel(int x,int y)
{
int scanleng=SCANLENG;  //扫描线所占字节数
int cur_page=CUR_PAGE;  //当前页
int color=CUR_COLOR.word;  //当前颜色
asm {
/*  以下14行，计算地址，换页  */
    mov ax,y
    mul scanleng
    shl x,1
    add ax,x
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov di,ax
```

```
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov ax,color
    mov es:[di],ax  //写显示存储器
    }
}

/*  高彩色模式下读象素点  */
COLOR VGAHIGH::getpixel(int x,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
union COLOR color={0L};
asm {
/*  以下14行，计算地址，换页  */
    mov ax,y
    mul scanleng
    shl x,1
    add ax,x
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov di,ax
    cmp dx,cur_page
    je jemp2
    }
    Select_Page(_DX);
jemp2:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov ax,es:[di]  //读显示存储器
    }
color.word=_AX;
return(color);
}
--------------------------------------------------------------------------------
```

7.5 扫描线操作

高彩色模式下的各种扫描线操作的步骤与256色模式非常类似，只不过高彩色模式下是以字为单位访问显示存储器，而256色模式是以字节为单位。

系统程序7-5 HIGH.CPP

```
--------------------------------------------------------------------------
/*  高彩色模式下画扫描线  */
void VGAHIGH::scanline(int x1,int x2,int y)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
int K;
int color=CUR_COLOR.word;
asm {
/*  以下13行，使x1<x2；计算扫描线的字节数(减1)，入栈*/
    shl x1,1
    shl x2,1
    mov ax,x2
    sub ax,x1
    jnc jemp1
    not ax
    inc ax
    mov bx,x2
    mov x1,bx
    }
jemp1:
asm {
    push ax
/*  以下14行，计算扫描线始点地址，换页  */
    mov ax,y
    mul scanleng
    add ax,x1
    jnc jemp2
    inc dx
    }
jemp2:
asm {
    mov di,ax  //始点的偏移地址存于di
    cmp dx,cur_page
    je jemp3
    mov cur_page,dx
    }
    Select_Page(cur_page);
jemp3:
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov ax,di
```

```
    pop cx  //扫描线字节数(减1)出栈
    add ax,cx
    jc jemp4  //扫描线跨页，转到jemp4
/*  以下6行，为扫描线不跨页时的操作  */
    shr cx,1
    inc cx
    mov ax,color
    cld
    rep stosw
    jmp end
    }
jemp4:
/*  以下为扫描线跨页时的操作  */
asm {
    push ax  //扫描线在下页的字节数(减1)入栈
    sub cx,ax
    shr cx,1
    mov ax,color
    cld
    rep stosw  //画出上页的扫描线
    inc cur_page
    }
    Select_Page(cur_page);  //将显示存储器换到下页
asm {
    mov ax,G_SEGMENT
    mov es,ax
    pop cx  //扫描线在下页的字节数(减1)出栈
    shr cx,1
    inc cx
    mov di,0h
    mov ax,color
    cld
    rep stosw  //画出下页的扫描线
    }
end:;
}

/*  高彩色模式下读扫描线  */
void VGAHIGH::getscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
int K1,K2;
asm {
/*  以下15行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    shl x1,1
```

```
    add ax,x1
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov si,ax
    cmp dx,cur_page
    je jemp2
    mov cur_page,dx
    }
    Select_Page(cur_page);
jemp2:
asm {
    mov ax,si
    mov cx,n
    shl cx,1
    dec cx
    add ax,cx
    jc jemp3  //若扫描线跨页，转到jemp3
    push ds
/*  以下6行，为扫描线不跨页时的操作  */
    inc cx
    les di,buf
    mov ax,G_SEGMENT
    mov ds,ax
    cld
    rep movsb  //读出整条扫描线
    pop ds
    jmp end
    }
jemp3:
/*  以下为扫描线跨页时的操作  */
asm {
    push ds
    sub cx,ax
    mov K1,ax
    les di,buf
    mov ax,G_SEGMENT
    mov ds,ax
    cld
    rep movsb  //读出上页的扫描线
    mov K2,di
    pop ds
    inc cur_page
    }
    Select_Page(cur_page);  //将显示存储器转到下页
asm {
```

```
    push ds
    mov cx,K1
    inc cx
    les di,buf
    mov di,K2
    mov ax,G_SEGMENT
    mov ds,ax
    mov si,0h
    cld
    rep movsb  //画出下页的扫描线
    pop ds
    }
end:;
}

/*  取高彩色模式下保存一条扫描线所需的缓冲区长度  */
int VGAHIGH::scanlinesize(int x1,int x2)
{
int K;
if(x1>x2)
    { K=x1; x1=x2; x2=K; }
K=(x2-x1+1)*2;
return K;
}

/*  高彩色模式下写扫描线*/
void VGAHIGH::putscanline(int x1,int y,int n,void *buf)
{
int scanleng=SCANLENG;
int cur_page=CUR_PAGE;
int K1,K2;
asm {
/*  以下15行，计算扫描线始点的地址，换页  */
    mov ax,y
    mul scanleng
    shl x1,1
    add ax,x1
    jnc jemp1
    inc dx
    }
jemp1:
asm {
    mov di,ax
    cmp dx,cur_page
    je jemp2
    mov cur_page,dx
    }
    Select_Page(cur_page);
```

```
jemp2:
asm {
    mov ax,di
    mov cx,n
    shl cx,1
    dec cx
    add ax,cx
    jc jemp3  //若扫描线跨页，转到jemp3
    push ds
/*  以下6行，为扫描线不跨页时的操作  */
    inc cx
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf
    cld
    rep movsb  //写入整个扫描线
    pop ds
    jmp end
    }
jemp3:
/*  以下为扫描线跨页时的操作  */
asm {
    push ds
    mov K1,ax
    sub cx,ax
    mov ax,G_SEGMENT
    mov es,ax
    lds si,buf
    cld
    rep movsb  //写入上页的扫描线
    mov K2,si
    pop ds
    inc cur_page
    }
    Select_Page(cur_page);  //将显示存储器换到下页
asm {
    push ds
    mov cx,K1
    inc cx
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0h
    lds si,buf
    mov si,K2
    cld
    rep movsb  //写入下页的扫描线
    pop ds
    }
```

```
end:;
}

/*  高彩色模式下置色清屏  */
void VGAHIGH::cls()
{
int col=CUR_COLOR.word;
int pn=PAGEN;  //所占用的显示存储器页数

asm mov si,0
loop:
Select_Page(_SI);
asm {
    mov ax,G_SEGMENT
    mov es,ax
    mov di,0h
    mov ax,col
    mov cx,8000h
    cld
    rep stosw

    inc si
    cmp si,pn
    jne loop
    }
}
```

7.6　参 数 设 置

系统程序7-6　HIGH.CPP

```
/*  320×200×32K色参数设置  */
_320_200_32K::_320_200_32K()
{
SCANLENG=640;  //每条扫描线所占字节数，其等于水平象素数乘2
WIDE=320;  //屏幕水平象素数
HIGH=200;  //屏幕垂直象素数
PAGEN=2;  //所占用的存储器页数
VESAmodeNo=0x010d;  //VESA模式号
}

/*  512×480×32K色参数设置  */
_512_480_32K::_512_480_32K()
{
SCANLENG=1024;
WIDE=512;
HIGH=480;
```

```
PAGEN=8;
VESAmodeNo=0x0170;
}

/*  640×480×32K色参数设置  */
_640_480_32K::_640_480_32K()
{
SCANLENG=1280;
WIDE=640;
HIGH=480;
PAGEN=10;
VESAmodeNo=0x0110;
}

/*  800×600×32K色参数设置  */
_800_600_32K::_800_600_32K()
{
SCANLENG=1600;
WIDE=800;
HIGH=600;
PAGEN=15;
VESAmodeNo=0x0113;
}

/*  320×200×64K色参数设置  */
_320_200_64K::_320_200_64K()
{
SCANLENG=640;
WIDE=320;
HIGH=200;
PAGEN=2;
VESAmodeNo=0x010e;
}

/*  512×480×64K色参数设置  */
_512_480_64K::_512_480_64K()
{
SCANLENG=1024;
WIDE=512;
HIGH=480;
PAGEN=8;
VESAmodeNo=0x0171;
}

/*  640×480×64K参数设置  */
_640_480_64K::640_480_64K()
{
SCANLENG=1280;
```

```
WIDE=640;
HIGH=480;
PAGEN=10;
VESAmodeNo=0x0111;
}

/*  800×600×64K色参数设置  */
_800_600_64K::_800_600_64K()
{
SCANLENG=1600;
WIDE=800;
HIGH=600;
PAGEN=15;
VESAmodeNo=0x0114;
}
--------------------------------------------------------------------------------
```

第 8 章　扩展内存(XMS)操作技术

下面各章所要介绍的某些技术和程序，将要涉及到大容量的数据操作问题，如绘图程序中的块操作、字符显示中的汉字字库的操作、图形打印中的图象缓存区的操作，这些操作中的数据量最大都可达到1M左右，显然不能将这样大量的数据放在常规内存中，放在硬盘中则速度较慢，所以在这些操作中都使用了扩展内存来存放数据。因此这里插入一章专门介绍扩展内存操作技术，并实现一组扩展内存操作的基础程序。

8.1　概　　述

8.1.1　PC机的内存类型

由于发展中的原因，目前，在DOS操作系统下运行的程序可以访问的PC机内存被分为以下五种。

1．常规内存(Conventional Memoey)

最前面的640K内存。这是DOS所能直接控制的唯一的一块内存，也是DOS下的应用程序能直接使用的唯一内存，应用程序对其它内存的访问都需要显式地通过一些专门方式来实现。应用程序并不能完全拥有常规内存，因为DOS内核、设备驱动程序、系统数据及一些内存驻留都需要使用这块内存。

最早的PC机只给应用程序提供了这640K的内存，DOS全部使用了它，而286以后的PC机所提供的内存就远远超出了这个数量，但由于兼容性方面的原因，DOS所直接控制的内存却始终没有突破这640K。

2．上位内存(Upper Memory BlocK，UMB)

640K以上1M以下的384K内存。这部分内存被保留供视频系统、ROM BIOS使用，但它并没有被完全使用，剩下的一部分就可以被应用程序使用。DOS下的应用程序可直接寻址这块内存，但DOS内核没有提供对这块内存的管理，因此只有通过一些专门的管理程序才能够有效地使用这块内存。从DOS 5.0开始，DOS提供了一种方式，能将设备驱动程序和内存驻留程序装载于这块内存，从而给应用程序留下了更多的常规内存。

3．扩展内存(Extended Memory)

1M以上的内存。286及其以上的PC机都可以拥有扩展内存，而它们大都已经拥有了扩展内存，但只有工作于保护模式下的程序才能直接访问扩展内存，而DOS工作于实模式，虽然也可以在DOS下编写工作于保护模式的程序，但这项工作是比较复杂和困难的，通常的DOS应用程序就只能通过一些扩展内存的管理程序来访问扩展内存。对扩展内存进行管理的方法有多种，目前最主要的一种是扩展内存管理规范XMS(eXtended Memory Specification)。

这一章所要介绍的就是扩展内存的操作技术。

4．扩充内存(Expanded Memory)

这是一种特殊的内存，它不是直接安装在主板的内存槽中，而是安装在一块插板上，这非常类似于显示卡中的显示存储器，所以严格地说，扩充内存应该是一种外部存储设备，但它具有与内存相当的速度。扩充内存的使用与CPU的寻址能力无关，不论是在实模式下还是

在保护模式下，CPU都不可能对扩充内存直接寻址。访问扩充内存的方式与显示存储器的换页访问方式几乎完全一样，这种内存有时也被称为扩页内存。Lotus、Inter、Microsoft三家公司共同制定了扩充内存管理规范EMS(Expanded Memory Specification)，现在的PC机已很少再安装扩充内存，但EMS却被用来作为管理扩展内存的一种方法。

在很多地方扩展内存和扩充内存的叫法正好颠倒，用扩页内存或用XMS、EMS来表示它们，可以避免这种混乱。

5．高区内存(High Memory Area，HMA)

扩展内存的前64K(实际为65520字节)。这是扩展内存中一块特殊的内存，它产生于如下原因：在实模式下，内存地址的表示方式为****H:****H，前面为段地址，后面为偏移地址，段地址乘16加偏移地址即得到实际地址，按照这种表示方式，能够表示出比1M多出65520个字节的地址，多出部分的范围为：0FFFFh:0010h～0FFFFh:0FFFFh，这些多出的部分在换算成实际地址时要用到第21位，而PC机在实模式下关断了第21根地址线(A20线)，从而使这些多出的地址又回绕到内存的最低端，但如果能够开启A20线，而在实模式下确实能够做到这一点，那么就能直接访问这多出的65520个字节。

在实模式下可直接访问高区内存，但这需要通过专门的操作才能实现，而且需要专门的程序对这块内存的使用进行管理。从DOS 5.0开始，在缺省设置下，DOS都将它本身的大部分内容装载于高区内存，以节省对常规内存的占用。

8.1.2　扩展内存使用方法

工作于实模式下的DOS应用程序不能直接使用扩展内存，这些程序必须采用专门的方法来使用扩展内存，这里介绍所能采用的一些方法。

1．INT 15功能

这是出现于PC/AT机的一个BIOS中断功能。它提供了一种很简陋的使用扩展内存的方法，它只具有两个功能，一是查询扩展内存的大小，二是在扩展内存和常规内存间传送数据，它没有提供对扩展内存进行管理的功能，当两个程序同时使用扩展内存时，几乎总是会发生冲突。这种方法在目前已很少使用。这种方法是由ROM BIOS所支持的，因此不需要安装任何驱动程序就可以使用它。

2．虚拟磁盘

这种方式将扩展内存虚拟成一个磁盘驱动器，其操作方式与通常的磁盘操作完全一样。虽然它比真实磁盘的速度要快得多，但与其它扩展内存使用方法相比，其速度则显得很慢，此外，虚拟盘的盘符是不固定的，这也会给编程带来麻烦。这种方法的最大好处就是它能被最终用户直接使用，实际上这种方法主要也是由最终用户在使用，而在编程中较少使用它。在DOS 3.*中提供了虚拟磁盘驱动程序VDISK.SYS，在以后的DOS中其改为RAMDRIVE.SYS。

3．EMS

这原本是用作扩充内存管理的一种方法，为了使那些使用扩充内存的程序也能在只装有扩展内存的机器上运行，这一方法被引入到扩展内存的管理中。在这种方式下，扩展内存中的某一块被映射到1M以下的某一段地址空间上，通过对这段地址进行读写操作，即可访问到相应的扩展内存块，通过变换映射到这一地址空间的扩展内存块，即可实现对全部扩展内存的访问。程序每次访问这段地址时，EMS管理程序都要在实模式和保护模式间进行切换，以

将这种访问导向扩展内存。从DOS 5.0开始，提供了EMS扩展内存管理程序，其为EMM386.SYS或EMM386.EXE，这一管理程序还提供了将设备驱动程序和内存驻留程序装载于UMB的功能。EMS是在应用程序中用得较多的一种扩展内存使用方法。

4. XMS

这是目前用得最多的一种扩展内存使用方法，DOS及Windows的设备驱动程序HIMEM.SYS实现了XMS管理功能，从DOS 5.0开始，HIMEM.SYS被作为一个缺省的设备驱动程序包含于系统配置文件CONFIG.SYS中，XMS扩展内存管理方法在目前已得到了非常普遍的支持。本书所要介绍的就是这种扩展内存使用方法。

8.1.3 XMS操作概述

XMS提供了对扩展内存，上位内存，高区内存进行管理的功能，在XMS中，扩展内存被称为EMB(Extended Memory BlocK)，且EMB不包含HMA，它们被当作两块独立的内存来看待。

通过系统配置文件安装了XMS驱动程序程序之后，系统中将出现两个中断调用功能，一个用于检测XMS管理程序的是否存在，一个用于获得XMS管理程序的地址。这两个功能的中断向量号为2Fh。下面说明这两个功能的调用格式。

(1)XMS检测功能

输入参数：AX＝4300h

返回值：AL＝80h　XMS管理程序存在

(2)XMS地址获取功能

输入参数：AX＝4310h

返回值：BX＝XMS管理程序的偏移地址

ES＝XMS管理程序的段地址

取得XMS的地址之后，通过对这一地址执行远调用，来使用XMS所提供的各项功能，调用时的输入参数及返回参数都放在CPU的有关寄存器中。XMS管理程序提供了18项功能，下节将说明这些功能的作用及调用格式。

8.2 XMS功能详解

XMS功能分为五组：功能0为管理程序信息；功能1～功能2为HMA管理；功能3～功能7为A20线控制；功能8～功能0Fh为EMB管理；功能10h～功能11h为UMB管理。

在调用XMS功能时，总是用AH来存放功能号。在返回值中，一般用AX来存放该次功能调同成功与否的状态，AX＝0表示失败，AX＝1表示成功，BL则一般用来存放错误代码。

功能0：取XMS版本号

输入参数：AH＝0

返回值：AX＝XMS版本号

BX＝XMS驱动程序内部版本号

DX＝HMA存在标志

0　HMA不存在

1　HMA存在

返回的版本号是16位BCD码值，如AX中的值为200h意味着管理程序支持的是XMS 2.00版

本。驱动程序内部编号主要用于调试。

功能1：请求高区内存

输入参数：AH＝1

DX＝请求长度，以字节表示

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

90h　不存在HMA

91h　HMA正被占用

92h　请求长度小于分配HMA的最小值

在XMS中HMA只能同时被一个程序使用，因此它设置了一个最小值，当请求长度小于该最小值时，XMS将认为不值得让该请求占用HMA，从而导致失败。该最小值在装载HIMEM.SYS时设置，如果没有设置，XMS将把HMA分配给第一个请求它的程序，而不管该程序请求的长度有多短。

功能2：释放高区内存

输入参数：AH＝2

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

90h　不存在HMA

93h　HMA未被分配

成功地请求了HMA的程序在退出前应释放HMA。HMA释放后，其不能再被访问，其中的数据和代码不再有效。

功能3：全程开启A20线

输入参数：AH＝3

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

82h　A20线出现错误

本功能只能由控制HMA的程序调用。如果开启成功，在程序退出之前应调用功能4关闭A20线。

功能4：全程关闭A20线

输入参数：AH＝4

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h 存在虚拟磁盘
82h A20线出现错误
94h A20线仍然可用

功能5：局部开启A20线

输入参数：AH＝5
返回值：AX＝状态
BL＝错误代码
80h 功能未实现
81h 存在虚拟磁盘
82h A20出现错误

本功能只应有需直接访问HMA的程序使用。如果开启成功，在程序退出之前应调用功能6来关闭A20线。

功能6：局部关闭A20线

输入参数：AH＝6
返回值：AX＝状态
BL＝错误代码
80h 功能未实现
81h 存在虚拟磁盘
82h A20线出现错误
94h A20线仍然可用

功能7：查询A20线状态

输入参数：AH＝7
返回值：AX＝状态
BL＝错误代码
0 无错
80h 功能未实现
81h 存在虚拟磁盘

功能8：查询自由的扩展内存

输入参数：AH＝8
返回值：AX＝以K字节计的最大自由扩展内存块的大小
DX＝以K字节计的自由扩展内存总的大小
BL＝错误代码
80h 功能未实现
81h 存在虚拟磁盘
0A0h 所有扩展内存都已被分配

返回值的大小不包括HMA。

功能9：分配扩展内存块

输入参数：AH＝9
DX＝以K字节计的内存块大小

返回值：AX＝状态

DX＝所分配块的句柄

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

0A0h　所有扩展内存都已分配

0A1h　所有扩展内存句柄均被使用

DX中所返回的句柄将在后继扩展内存管理功能中使用。若分配失败，返回的句柄为空。若分配成功，程序在退出前应调用0Ah释放所分配到的内存。

功能0Ah：释放扩展内存块

输入参数：AH＝0Ah

DX＝所要释放的内存块的句柄

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

0A2h　句柄无效

0ABh　句柄被加锁

功能0Bh：移动内存块

输入参数：AH＝0Bh

DS:SI＝内存移动信息块的指针，该信息块的结构见表3-1。

表3-1　内存移动信息块结构

字　节　数	含　　义
4	所要移动的内存块的大小，以字节为单位
2	源块句柄
4	所移动的块在源块中的偏移值，以字节为单位
2	目的块句柄
4	所移动的块移到目的块中的偏移值，以字节为单位

返回值：AX＝状态

BL＝错误代码

80h　功能未实现

81h　存在虚拟磁盘

0A3h　无效源句柄

0A4h　无效源偏移量

0A5h　无效目的句柄

0A6h　无效目的偏移量

0A7h　长度无效

0A8h　移动有重叠

0A9h 奇偶校验错

这是对扩展内存操作是最常用的一个功能。

本功能主要用于在常规内存和扩展内存之间移动数据块，但它也可在扩展内存内或常规内存内移动数据块。

所移动块的长度必须为偶数。

移动时源块和目的块都必须未被加锁。

如果块重叠，则只能向前移动，即目的地址必须小于源块地址。

如果源块或目的块在常规内存中，这时它的句柄为0，偏移值用实模式下的标准地址表示方式表示，即用段地址:偏移地址来表示。

功能0Ch：扩展内存块加锁

输入参数：AH＝0Ch

DX＝要加锁的内存块的句柄

返回值：AX＝状态

DX＝内存块的32位绝对地址的高字节

BX＝内存块的32位绝对地址的低字节

BL＝错误代码

80h 功能未实现

81h 存在虚拟磁盘

0A2h 句柄无效

0ACh 块的加锁计数溢出

0ADh 加锁失败

本功能用于禁止移动某一内存块，并获得其绝对地址。只有当加锁成功时，返回的地址才有效。被加锁的块应尽快解锁。每个已分配的扩展内存块都有一个加锁计数器，对扩展内存块进行一次加锁，其加锁计数器就加1。

功能0Dh：扩展内存块解锁

输入参数：AH＝0Dh

DX＝所要解锁的内存块的句柄

返回值：AX＝状态

BL＝错误代码

80h 功能未实现

81h 存在虚拟磁盘

0A2h 句柄无效

0AAh 块未加锁

调用一次本功能，相应扩展内存块的加锁计数器就减1，只有加锁计数器为0时，扩展内存块才被真正解锁，这时对该块的进一步解锁将无效。

功能0Eh：取扩展内存块信息

输入参数：AH＝0Eh

DX=块句柄

返回值：AX＝状态

BH＝块的加锁计数器值
BL＝如果成功　系统中还未用的扩展内存句柄数
　　如果失败　错误代码
　　80h　功能未实现
　　81h　存在虚拟磁盘
　　0A2h　句柄无效

功能0Fh：重新分配扩展内存

输入参数：AH＝0Fh
　　BX＝以K字节计的块的新的大小
　　DX＝要改变大小的块的句柄
返回值：AX＝状态
　　BL＝错误代码
　　80h　功能未实现
　　81h　存在虚拟磁盘
　　0A0h　所有扩展内存以分配
　　0A2h　句柄无效
　　0ABh　块被加锁

如果新块大小小于原块，在原块高端的数据将丢失。

功能10h：分配上位内存块

输入参数：AH＝10h
　　DX＝以节(16字节)计的块的大小
返回值：AX＝状态
　　BX＝如果成功　所分配块的段地址
　　BL＝如果失败　错误代码
　　80h　功能未实现
　　0B0h　一个较小的上位内存块可用
　　0B1h　没有可用的上位内存
　　DX＝如果成功　已分配块的实际节数
　　如果失败　最大可用块的节数

所分配的上位内存块总是节对准的，因此只需返回它的段地址，上位内存在1M以内，用所返回的段地址即可直接对所分配的上位内存块寻址。

功能11h：释放上位内存块

输入参数：AH＝11h
　　DX＝所要释放内存块的段地址
返回值：AX＝状态
　　BL＝错误代码
　　80h　功能未实现
　　0B2h　段地址无效

8.3 程 序 设 计

8.3.1 功能选择

XMS提供了使用EMB、UMB、HMA三种内存的功能，但对应用程序来说，并不是每一种内存都有实际的使用价值，下面分别讨论。

1．HMA

HMA大小非常有限，只有64K，而且在目前的大部分PC机中，都将DOS=HIGH放在了系统配置文件中，也就是说在大部分PC机上，HMA都始终由DOS本身占据着，应用程序根本不可能请求到HMA。如果试图用某种方法使程序所运行于的PC机不指定DOS=HIGH，那也只会是一种得不偿失的做法，因为这时DOS会让你失去一块相当大的常规内存(约46K)，而常规内存的使用更加自如，更加高效。所以，应用程序利用HMA的最好方法就是根本不去使用它，把它让给DOS，这样你就能轻松地获得更大的内存，而且是常规内存。

2．UMB

在整个384K的UMB中，视频地址占用了128K，视频BIOS约占用32K，其它BIOS约占用64K～128K，最终留给应用程序的UMB约为96K～160K，这还是具有一定的使用价值。但为了能将设备驱动程序及内存驻留程序装载于UMB，在很多PC机上，都安装了EMM386.EXE并指定了DOS=UMB，这时UMB将全部由DOS来控制，XMS驱动程序将完全失去对UMB的控制权，这时应用程序就无法通过XMS来操作UMB。所以在应用程序中一般不要通过XMS来使用UMB，将UMB留给DOS去装载设备驱动程序及内存驻留程序，以空出更多的常规内存，这是利用UMB的最好方式。

当UMB的控制权移交给DOS后，DOS按控制常规内存的方式，对UMB进行了完全的控制，一个能在常规内存中运行的程序(包括非内存驻留程序)，同样也能在UMB中运行，只要UMB的空间足够。但UMB和常规内存没有链接在同一条内存控制链上，它们拥有各自的内存控制链，所以这两块内存仍然是分离的，按通常的内存申请方式，在UMB中运行的程序无法使用常规内存的空间，同样在常规内存中运行的程序也无法获得UMB中的空间。一个程序是装载于常规内存还是装载于UMB，这通常由最终用户来决定，程序本身不必考虑这个问题。

3．EMB

这是由XMS所控制的最大一块内存，目前的大多数PC机一般都装有4M内存，其中的3M即为扩展内存，这是HMA和UMB所无法比拟的。当然DOS不会把它们都留给应用程序，DOS的一些设备驱动程序会占用这部分内存，在最不利的情况下，4M的PC机中，留给应用程序的自由扩展内存仍将有1.5M。显然扩展内存是XMS所提供的最有价值的东西，获得扩展内存是大多数情况下使用XMS的唯一目的。

由以上讨论可见，XMS所提供的UMB管理功能、HMA管理功能及相应的A20线控制功能，在通常的应用程序中极少有可能用到，此外EMB管理中的内存块加锁、解锁功能也同样极少用到，它们通常在保护模式的编程中使用。对这些极少用到的功能，将不在编程中实现它们，在编程中将只实现那些常用的扩展内存管理功能，包括：查询自由的扩展内存，分配扩展内存块，释放扩展内存块，移动扩展内存块，重新分配扩展内存块。

8.3.2　对程序功能的处理

由驱动程序所提供的原始XMS操作功能，其结构和含义都已经比较清晰和明确，但在程序中还是需要对某些功能作如下一些处理：

(1)移动扩展内存块将是在应用程序中最常用的一项操作，在应用程序中这一操作将明显分为两种：一是将常规内存的数据块移到扩展内存，即写扩展内存；二是，将扩展内存的数据块移到常规内存，即读扩展内存。所以，程序中将内存块移动功能分为了两个功能：读扩展内存、写扩展内存，而原来的移动功能将被隐藏起来。

(2)在移动内存块时，块的长度必须是偶数。在程序中将要解决这一问题，即所实现的扩展内存读、写功能要支持应用程序对扩展内存进行奇数长度的读写。

(3)在程序中将查询自由扩展内存功能，分为取最大自由扩展内存块的大小和取自由扩展内存总的大小两项功能，这样更便于应用程序使用。

8.3.3　编程方案

要实现的所有功能都包含在一个类中，该类具有两个数据成员和若干个成员函数，各项XMS功能由该类的各个成员函数来提供，每个成员函数提供一种功能。该类的一个对象对应于应用程序所申请的一块扩展内存。该类的某些功能是针对于整个扩展内存的，它们与具体的某个扩展内存块无关，实现这些功能的函数都将设置成静态成员函数。

以下给出该类的说明。

系统程序8-1　XMS.H

```
--------------------------------------------------------------------------
class XMS {
    int move(struct EMB *emb);  //移动内存块，私有成员
public:
    static int OK;  //XMS扩展内存是否存在的标志
    int handle;  //扩展内存块的句柄

    XMS(int size);  //构造函数，分配扩展内存
    ~XMS();  //析构函数，释放扩展内存
    static int init(void);  //初始化
    static unsigned freesize(void);  //取自由扩展内存总的大小
    static unsigned largestblocK(void);  //取最大自由扩展内存块的大小
    int realloc(int size);  //重新分配扩展内存
    int put(void *dp,void *sp,long leng);  //写扩展内存
    int get(void *dp,void *sp,long leng);  //读扩展内存
    };
--------------------------------------------------------------------------
```

8.4　程　　序

系统程序8-2　XMS.CPP

```
--------------------------------------------------------------------------
#include <dos.h>
```

```
#include <alloc.h>
#include <mem.h>
#include "\vga\xms\xms.h"

/*  内存移动信息块结构  */
struct EMB {
    long Leng;  //所要移动的内存块的大小，以字节为单位
    unsigned SourceHandle;  //源块句柄
    long SourceOfs;  //所移动的块在源块中的偏移值，以字节为单位
    unsigned DestinHandle;  //目的块句柄
    long DestinOfs;  //所移动的块移到目的块中的偏移值，以字节为单位
    };

int XMS::OK=0;  //0表示XMS管理程序不存在，1表示存在
static void far *XMSaddr;  //存放系统中XMS管理程序的地址，见程序说明①

/*************************************************************************
功能：判别XMS驱动程序是否存在，如存在，取XMS管理程序的地址
输入参数：无
返回值：XMS驱动程序是否存在的标志，该标志同时存于XMS::OK中
        0  XMS驱动程序不存在
        1  XMS驱动程序存在
说明：在应用程序中必须首先运行此函数，然后才能够使用该类的其它函数
*************************************************************************/
int XMS::init(void)
{
static struct REGPACK rg;

rg.r_ax=0x4300;
intr(0x2f,&rg);
if( (rg.r_ax&0x00ff) == 0x80 )
    {
    rg.r_ax=0x4310;
    intr(0x2f,&rg);
    XMSaddr=MK_FP(rg.r_es,rg.r_bx);
    OK=1;
    }
else
    OK=0;
return(OK);
}

/*************************************************************************
功能：取当前自由扩展内存总的大小
输入参数：无
返回值：以K字节计的自由扩展内存的大小
*************************************************************************/
unsigned XMS::freesize(void)
```

```
{
if(OK==0)
    return(0);
asm {
    mov ah,8
    call XMSaddr
    }
return _DX;
}

/*************************************************************************
功能：取当前最大的扩展内存块的大小
输入参数：无
返回值：以K字节计的最大扩展内存块的大小
*************************************************************************/
unsigned XMS::largestblocK(void)
{
if(OK==0)
    return(0);
asm {
    mov ah,8
    call XMSaddr
    }
return _AX;
}

/*************************************************************************
功能：分配一块扩展内存
输入参数：KSize＝所要分配的扩展内存块的大小，以K字节为单位
返回值：所分配的扩展内存块的句柄。若该值为0，表示分配失败
*************************************************************************/
XMS::XMS(int size)
{
if(OK==0)
    {
    handle=0;
    return;
    }

asm {
    mov ah,9
    mov dx,size
    call XMSaddr
    }
handle=_DX;  //若handle==0，表示分配失败，该值可作为分配是否成功的标志
}

/*************************************************************************
```

```
功能：析构函数，释放扩展内存块
输入参数：Handle＝所要释放的扩展内存的句柄
返回值：状态：0失败，1成功
*************************************************************************/
XMS::~XMS()
{
if(handle==0)
    return;

int hd=handle;  //见程序说明①
asm {
    mov ah,0ah
    mov dx,hd
    call XMSaddr
    }
}

/*************************************************************************
功能：以新的尺寸重新分配扩展内存
输入参数：KSize＝新的大小
          Handle＝要重新分配的扩展内存块的句柄
返回值：状态：0失败，1成功
*************************************************************************/
int XMS::realloc(int size)
{
if(handle==0)
    return(0);

int hd=handle;  //见程序说明①
asm {
    mov ah,0fh
    mov bx,size
    mov dx,hd
    call XMSaddr
    }
return _AX;
}

/*************************************************************************
功能：移动内存块
输入参数：内存移动信息块结构的指针
返回值：状态：0失败，1成功
    该成员函数作为私有成员被隐藏起来，在应用程序中它是不可见的
*************************************************************************/
int XMS::move(struct EMB *emb)
{
asm {
    push ds
```

```
    mov ah,0bh
    push ds
    pop es
    lds si,emb
    call es:XMSaddr
    pop ds
    }
return _AX;
}

/*************************************************************************
功能：写扩展内存
输入参数：dp＝所写入数据块在目的块中的偏移值
          sp＝数据块在常规内存中的地址
          Count＝所写入数据块的长度，以字节为单位
返回值：状态：0失败，1成功
    源块只能在常规内存中，其句柄恒为0，因此不需传入源块的句柄。该函数负责对奇数数据长度的
处理。
*************************************************************************/
int XMS::put(void *dp,void *sp,long leng)
{
struct EMB emb;

if(leng&1L)  //长度为奇数
    leng++;  //使长度为偶数，见程序说明②
emb.Leng=leng;
emb.SourceHandle=0;
emb.SourceOfs=(long)sp;
emb.DestinHandle=handle;
emb.DestinOfs=(long)dp;
return move(&emb);
}

/*************************************************************************
功能：读扩展内存
输入参数：dp＝存放所读数据的常规内存缓冲区的地址
          sp＝所读数据在源块中的偏移
          leng＝所读数据的长度，以字节计
返回值：状态：0失败，1成功
说明：目的块只能在常规内存中，其句柄恒为0，因此不需传入目的块的句柄。该函数负责对奇数数据
长度的处理。尽管该函数能读奇数长度的数据，但在长度为奇数时其速度明显较慢。
*************************************************************************/
int XMS::get(void *dp,void *sp,long leng)
    {
    int v;
    struct EMB emb;

    if(leng&1L)  //长度为奇数
```

```
    {
    char *p,*d;
    leng--;
    if( leng>0 )
        {
        emb.Leng=leng;
        emb.SourceHandle=handle;
        emb.SourceOfs=(long)sp;
        emb.DestinHandle=0;
        emb.DestinOfs=(long)dp;
        move(&emb);
        }

    p=(char *)malloc(2);
    emb.Leng=2L;
    emb.SourceHandle=handle;
    emb.SourceOfs=(long)sp+leng;
    emb.DestinHandle=0;
    emb.DestinOfs=(long)p;
    v=move(&emb);
    d=(char *)dp;
    d[leng]=p[0];
    free(p);
    }
else  //长度为偶数
    {
    emb.Leng=leng;
    emb.SourceHandle=handle;
    emb.SourceOfs=(long)sp;
    emb.DestinHandle=0;
    emb.DestinOfs=(long)dp;
    v=move(&emb);
    }
return(v);
}
```

程序说明：

①C++中的几乎所有类型的数据变量都可作为嵌入汇编的操作数，但类的数据成员不能直接作为嵌入汇编的操作数。

②在这种偶数处理方式中，当将一块连续的数据分成多段写入扩展内存块，且先写后面的数据，这时如果后写的数据长度为奇数时，后写的数据将破坏先写数据的第1个字节，在应用程序中有两种方法可避免错误的发生：一是，从前到后顺序写入各段数据，而不要倒着写；二是，当必须倒着写时，将数据段的长度控制为偶数。因为扩展内存是按K字节分配的，所以在这种偶数处理方式中，不必担心所写入的数据会超出所分配的扩展内存空间。

8.5　程序使用方式

在应用程序中不能对扩展内存中的数据直接进行处理，从使用方式上看，扩展内存很类似于外部存储器：当需要将数据存放于扩展内存时，首先必须在常规内存中生成这些数据，然后再将它们写入扩展内存；当需要使用存储在扩展内存中的数据时，必须首先将数据从扩展内存读到常规内存，然后再使用它们。

在应用程序中使用扩展内存的通常步骤如下：

①运行XMS::init()。检测扩展内存驱动程序是否存在，若存在取得其地址；

②定义一个XMS类的对象，记为a。分配一块扩展内存，若a.handle==0，表示分配失败，对象a不可用；

③在常规内存中生成需保存在扩展内存中的数据；

④用a.put()，将数据写入扩展内存；

⑤用a.get()，将扩展内存中的数据读到常规内存；

⑥在常规内存中使用所读出的数据；

⑦若a是动态定义的，删除对象a，以释放扩展内存。

下面给出一个使用扩展内存的示例程序，该程序通过扩展内存交换两块常规内存中的数据。

示例程序8-1

```
void main()
{
char b1[2048],b2[2048];  //在常规内存中建立两个数据块
int i;
for(i=0;i<1024;i++)
    {
    b1[i]=1;
    b2[i]=2;
    }
XMS a1(2);  //定义一个XMS对象，分配一块2K的扩展内存
a1.put(0,b1,2048);  //将b1中的数据放入刚分配的扩展内存
XMS *a2;  //定义一个XMS对象的指针，此时不分配扩展内存
a2=new XMS(2);  //动态定义一个XMS对象，此时分配一块2K的扩展内存
a2->put(0,b2,2048);  //将b2中的数据放入刚分配的扩展内存
a1.get(b2,0,2048);  //将第1块扩展内存中的数据放到b2中
a2->get(b1,0,2048);  //将第2块扩展内存中的数据放到b1中
delete a2;  //删除动态分配的对象a2，以释放相应的扩展内存
  //a1在程序退出时将被自动删除
}
```

第 9 章　基本绘图功能

9.1　概　　述

除了画点、画水平直线之外，一个基础图形支持系统还必须具备绘制一些基本几何图形的功能，这些基本几何图形通常包括：直线、矩形、多边形、圆、椭圆、扇形、实面积等。本章将介绍这些功能的算法及程序。

这些几何图形通常用一定的几何参数和数学公式来描述，而在一个光栅图形系统中，任何一个图形都只能由一些离散的象素点来表示，因而绘制一个几何图形所要完成的主要工作就是，根据图形的几何参数及数学公式确定出构成该图形的所有象素点，这一工作通常称为扫描转换。

用数学公式表示的几何图形是一种抽象的图形，它由无穷多个无限小的连续的点所组成，而光栅图形系统中的象素点具有一定的大小，其数量是有限的，位置也是离散的，因此在光栅图形系统中只能近似地表示出一个几何图形，扫描转换的一个主要目标就是要使转换的结果尽量逼近原图形。

每种几何图形都有一定的数学公式，根据其数学公式并作一定的取整处理，似乎就可以很简单地完成扫描转换，但这样进行扫描转换的速度是很慢的，因为各种几何图形的数学公式中都包含了实数乘法运算，有的还包含了实数的除法、开方甚至三角函数运算，因此在扫描转换中一般不直接采用几何图形的原公式，而需要研究专门的算法。扫描转换一直是计算机图形学的一项基本内容，各种几何图形的扫描转换都已经有了一些很成熟的算法，本书直接采用了这些已有的算法，这些算法都避免了各种浮点运算，主要用整数的加减运算或乘法运算来实现扫描转换。

在本书的程序中，通过调用各种图形模式下的写点或画扫描线函数来显示扫描转换所得到的各个象素点，各个绘图函数不需要直接与VGA打交道，因此本书所实现的绘图函数与图形模式无关，所有的图形模式共用一组绘图函数。

9.2　画　直　线

9.2.1　算法

直线的扫描转换算法有好几种，这里所介绍和采用的是由Bresenham所提出的一种算法，后面的画圆、画椭圆所采用的也都是Bresenham的算法。

直线方程为：$y=mx+b$。一条具体的直线由两个端点的坐标(x_0,y_0)和(x_n,y_n)来确定，端点的坐标值都为整数。两个端点的坐标给定后直线方程中的两个参数可按下式求得：

$$m=(y_n-y_0)/(x_n-x_0)=dy/dx$$

$$b=y_0-mx_0=y_n-mx_n$$

有：$dy=y_n-y_0$，$dx=x_n-x_0$

直线扫描转换的任务就是确定出若干个象素点$P_0(x_0,y_0)$、…、$P_i(x_i,y_i)$、…、$P_n(x_n,y_n)$，用这些象素点来构成所给定的一条直线，各象素点的坐标值都为整数。

为简单起见，首先给出两个假设，在这两个假设的基础上来推导直线的扫描转换算法。

假设1：直线的斜率在[0,1]范围内，且$x_n>x_0$。

假设2：$x_0=0$，$y_0=0$，即直线的一个端点在坐标原点。

通过一定的坐标变换可将任何一条直线转换为符合上述两个假设。

由假设2，直线方程可表示为：$y=(dy/dx)x$

不失一般性设：$x_i=x_{i-1}+1$，由假设1，则必有$y_i \geq y_{i-1}$。

这里将采用递推方法进行直线的扫描转换。

假定已求得一个点$P_{i-1}(x_{i-1},y_{i-1})$，现在求下一个点$P_i(x_i,y_i)$，因$x_i=x_{i-1}+1$，所以实际只需求y_i。

由假设1，y_i的取值只可能有两种情况：$y_i=y_{i-1}+1$或$y_i=y_{i-1}$，这两种取值与y_i的理论取值的正偏差分别记为s和t，有：

$$s=y_{i-1}+1-(dy/dx)(x_{i-1}+1)$$

$$t=(dy/dx)(x_{i-1}+1)-y_{i-1}$$

有：

$$s-t=2y_{i-1}+1-2(x_{i-1}+1)(dy/dx)$$

令：

$$d_i=(s-t)dx$$

有：

$$d_i=dx-2dy+2(y_{i-1}dx-x_{i-1}dy) \qquad (9\text{-}1)$$

因dx>0，所以d_i的正负情况即反映了y_i的两种取值与其理论取值之间偏差的相对大小，称d_i为判别变量。若$d_i<0$，则$y_{i-1}+1$更接近于理论值，应取$y_i=y_{i-1}+1$；若$d_i>0$，则应取$y_i=y_{i-1}$；若$d_i=0$，则两种取值都可。

下面推导计算d_i的递推公式。由式9-1可得：

$$d_{i+1}=dx-2dy+2(y_idx-x_idy) \qquad (9\text{-}2)$$

式9-2减式9-1，并由$x_i=x_{i-1}+1$，可得：

$$d_{i+1}=d_i+2dx(y_i-y_{i-1})-2dy \qquad (9\text{-}3)$$

根据y_i的两种取值情况可对式9-3作进一步简化。当$d_i<0$时，可将$y_i=y_{i-1}+1$代入式中，当$d_i \geq 0$时，则可将$y_i=y_{i-1}$代入式中，由此将式9-3分为如下两个公式

$$d_{i+1}=d_i+2(dx-dy) \qquad \text{当}d_i<0 \qquad (9\text{-}4)$$

$$d_{i+1}=d_i-2dy \qquad \text{当}d_i \geq 0 \qquad (9\text{-}5)$$

当i=1时，由式9-1及假设2，可得：

$$d_1=dx-2dy \qquad (9\text{-}6)$$

式9-6、式9-4、式9-5即为求d_i的递推公式。

按照上述推导即可构造一套进行直线扫描转换的递推算法，下面给出按该算法对满足假设1的直线进行扫描转换的步骤。

①令$dx=x_n-x_0$，$dy=y_n-y_0$；

　令$x=x_0$，$y=y_0$，画出象素点(x,y)；

②令dst=dx-2dy；

③若dst<0，则：令dst=dst+2(dx-dy)，令y=y+1；

否则：令dst=dst-2dy；

④令x=x+1，画出象素点(x,y)；

⑤若$x \geq x_n$，则：结束；

否则：转到③

在编程中该算法还需进行优化。

经过适当的坐标转换可以使任何一条直线转换为满足假设1，从而使得这一算法可以适用于任意直线，具体的坐标转换方法这里不再介绍，下面给出的画线程序包含了对所有情况的处理。

9.2.2 画线程序

在程序中对水平直线直接调用画扫描函数画出，对垂直直线则采用了专门的算法，只对斜直线采用了Bresenham算法。

程序中实现了两个具有不同接口的画线函数，一是，直接给定两个端点的坐标值；二是，只给定末点的坐标值，始点的坐标值则为由程序系统所维持的画线开始点。直接实现的是第一个函数，第二函数通过调用第一个函数来实现。在程序中，画线开始点始终为最近一次画线操作的末点，也可通过一个专门的函数来设置画线开始点的位置。

系统程序9-1 VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：在两个点之间画一条直线
输入参数； x1＝第一个端点的横坐标
           y1＝第一个端点的纵坐标
           x2＝第二个端点的横坐标
           y2＝第二个端点的纵坐标
返回值：无
*************************************************************************/
void VGABASE::line(int x1,int y1,int x2,int y2)
{
int i;
int x;  //用作存放 dx=x2-x1，也用作临时变量
int y;  //用作存放 dy=y2-y1，也用作临时变量
int p;  //用于存放 2dy
int n;  //用于存放 2(dx-dy)
int tn; //判别变量

CUR_X=x2;  //画线开始点的横坐标值
CUR_Y=y2;  //画线开始点的纵坐标值
if( y1==y2 )  //如果为水平直线，直接调用画扫描线函数
    {
    if(x1>x2)
        { x=x2; x2=x1; x1=x; }
    scanline(x1,x2,y1);
    return;
    }
if( x1==x2 )  //如果为垂直直线，采用专门的方法
```

```
    {
    if(y1>y2)
        { y=y2; y2=y1; y1=y; }
    for(i=y1;i<=y2;i++)
        putpixel(x1,i);
    return;
    }

if( abs(y2-y1) <= abs(x2-x1) )  //斜率为[-1,1]
    {
    if( (y2<y1&&x2<x1) || (y1<=y2&&x1>x2) )
        {
        x=x2; y=y2; x2=x1; y2=y1; x1=x; y1=y;  //交换端点，使x1<x2
        }
    if( y2>=y1 && x2>=x1 )  //斜率为[0,1]
        {
        x=x2-x1; y=y2-y1;
        p=2*y; n=2*x-2*y; tn=x;
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1++; }
            putpixel(x1,y1);
            x1++;
            }
        }
    else  //斜率为[-1,0)
        {
        x=x2-x1; y=y2-y1;
        p=-2*y; n=2*x+2*y; tn=x;
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1--; }
            putpixel(x1,y1);
            x1++;
            }
        }
    }
else  //斜率为(1,∞)或(-∞,-1)
    {
    x=x1; x1=y2; y2=x; y=y1; y1=x2; x2=y;  //将两个端点以直线y=x为轴作反射变换
    if( (y2<y1&&x2<x1) || (y1<=y2&&x1>x2) )
        {
        x=x2; y=y2; x2=x1; y2=y1; x1=x; y1=y;  //使x1<x2
        }
    if( y2>=y1 && x2>=x1 )  //斜率为(1,∞)
        {
        x=x2-x1; y=y2-y1;
        p=2*y; n=2*x-2*y; tn=x;
```

```
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1++; }
            putpixel(y1,x1);
            x1++;
            }
        }
    else  //斜率为(-∞,-1)
        {
        x=x2-x1; y=y2-y1;
        p=-2*y; n=2*x+2*y; tn=x;
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1--; }
            putpixel(y1,x1);
            x1++;
            }
        }
    }
}

/*************************************************************************
功能：在画线开始点至给定点间画一条直线
输入参数：  x＝给定点的横坐标
            y＝给定点的纵坐标
返回值：无
*************************************************************************/
void VGABASE::lineto(int x,int y)
{
line(CUR_X,CUR_Y,x,y);
}

/*  设置画线开始点的位置  */
void VGABASE::moveto(int x,int y)
{
CUR_X=x;
CUR_Y=y;
}
-------------------------------------------------------------------------
```

9.2.3 画矩形及多边形

画矩形及画多边形的函数通过调用画直线函数来实现，因此它们非常简单。程序中实现了两个画多边形的函数，一个用于画一个多边形，另一个则用于一次画多个多边形，它们使用了同一个函数名，但具有不同的输入参数。这两个画多边形函数的输入参数的格式见示例程序9-1。

系统程序9-2　VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：画一个多边形
输入参数：  n＝多边形的顶点数
            border＝多边形各顶点的坐标
返回值：无
*************************************************************************/
void VGABASE::poly(int n,int *border)
{
int i,nn;
nn=n*2;
line(border[nn-2],border[nn-1],border[0],border[1]);
for(i=2;i<nn;i+=2)
    lineto(border[i],border[i+1]);
}

/*************************************************************************
功能：画多个多边形
输入参数：border＝各多边形的顶点数及各多边形各顶点的坐标
返回值：无
*************************************************************************/
void VGABASE::poly(int *border)
{
int i=0,n;
while(1) {
    n=border[i];i++;
    if(n==0)
        breaK;
    poly(n,border+i);  //通过调用画一个多边形的函数实现
    i += (2*n);
    }
}

/*************************************************************************
功能：画矩形
输入参数：  x1＝矩形左上角的横坐标值
            y1＝矩形左上角的纵坐标值
            x2＝矩形右下角的横坐标值
            y2＝矩形右下角的纵坐标值
返回值：无
*************************************************************************/
void VGABASE::rectangle(int x1,int y1,int x2,int y2)
{
line(x1,y1,x2,y1);
line(x2,y1,x2,y2);
line(x1,y2,x2,y2);
line(x1,y1,x1,y2);
```

```
}
--------------------------------------------------------------------------
```

示例程序9-1

```
void main()
{
_640_480_16 A;
A.init();
int *b1[10]={320,0,  480,80,  400,240,  240,240,  160,80};
  //一个五边形各顶点的坐标值，为：x1,y1,x2,y2,x3,y3,x4,y4,x5,y5
int *b2[]={3,  100,0, 150,100, 50,100,  3, 100,20, 130,80, 70,80,  0}
  //两个三角形的顶点数及各顶点的坐标值
A.setcolor(7);
A.poly(5,b1);  //画出一个五边形
A.poly(b2);  //画出两个三角形
getch();
A.close();
}
```

9.3 画圆、画扇形

9.3.1 画圆算法

由于圆所具有的对称性，不需要逐个计算圆周上的所有点，而只需计算45度至90度之间1/8圆周上的点，然后通过对称关系得到圆周上的其它点。对这1/8圆周进行扫描转换的方法与直线扫描转换非常类似，也是构造一个判别变量，通过该变量的符号来确定当x方向增1时，y方向是不变还是减1。判别变量的递推公式如下：

$$d_1=3-2R \tag{9-7}$$

$$d_{i+1}=d_i+4x_{i-1}+2 \quad \text{当} d_i<0 \tag{9-8}$$

$$d_{i+1}=d_i+4(x_{i-1}-y_{i-1})+6 \quad \text{当} d_i\geq 0 \tag{9-9}$$

其中：x_i,y_i为相对于圆心的偏移量，R为半径。

确定y_i取值的判别规则为：若$d_i<0$，则$y_i=y_{i-1}$；若$d_i\geq 0$，则$y_i=y_{i-1}-1$。

下面给出对圆进行扫描转换的Bresenham算法。

设圆心的坐标为(x_0,y_0)，圆的半径为R。

①令x=0，y=R，画出点(x_0+x,y_0+y)及该点在圆周上的其它7个对称点；

②令dst=3-2R;

③若$d_i<0$，则：令dst=dst+4x+2;

否则：令dst=dst+4(x-y)+6，令y=y-1；

④令x=x+1，画出点(x_0+x,y_0+y)及该点在圆周上的其它7个对称点；

⑤若x≥R×sin45，则：结束。

否则：转到③。

9.3.2　画圆程序

在程序中判别变量的值很容易超出2字节短整型数的取值范围，因此必须将判别变量定义为4字节的长整型数。

系统程序9-3　VGADRAW.CPP

```
--------------------------------------------------------------------------
static double SIN45=0.707106781186548;

/************************************************************************
功能：画圆
输入参数：  x0＝圆心的横坐标
            y0＝圆心的纵坐标
            r＝半径
************************************************************************/
void VGABASE::circle(int x0,int y0,int r)
{
int i;
long tn;  //判别变量
int x,y,xmax;

y=r; x=0;
xmax=(double)r*SIN45;
tn=(1-r*2);
while(x<=xmax) {
    if(tn>=0)
        {
        tn += ( 6 + ((x-y)<<2) );
        y--;
        }
    else
        tn += ( (x<<2) + 2 );
    putpixel(x0+y,y0+x); //0度到45度(屏幕坐标系，下同)
    putpixel(x0+x,y0+y); //45度到90度
    putpixel(x0-x,y0+y); //90度到135度
    putpixel(x0-y,y0+x); //135度到180度
    putpixel(x0-y,y0-x); //180度到225度
    putpixel(x0-x,y0-y); //225度到270度
    putpixel(x0+x,y0-y); //270度到315度
    putpixel(x0+y,y0-x); //315度到360度
    x++;
    }
}
--------------------------------------------------------------------------
```

9.3.3　画扇形

扇形是圆的一部分，这里在上述画圆算法的基础上构造了画扇形的算法。在算法中主要增加了一个判别步骤，来挑选出那些需要画出的点，具体处理方式见程序。在程序中，用起

始角度、终止角度及一个圆来定义一个扇形，扇形从起始角度开始按逆时针方向画至终止角度，起始角度和终止角度按正常坐标系(向上为Y轴的正方向)定义，而不是按屏幕坐标系(向下为Y轴的正方向)定义。

系统程序9-4 VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：画扇形
输入参数： x0＝圆心的横坐标
           y0＝圆心的纵坐标
           r＝半径
           stangle＝扇形的起始角度(0～360)
           endangle＝扇形的终止角度(0～360)
返回值：无
*************************************************************************/
void VGABASE::sector(int x0,int y0,int r,int stangle,int endangle)
{
int i,j;
int *xy;
int bx,ex,bxd,exd,bxf,exf,ben;
long tn,x,y;
long xmax;

y=r; x=0;
xmax=(double)r*SIN45;
tn=(1-r*2);

xy=(int *)calloc(20,2);  //存放圆周上的8个对称点
xy[0]=x0+r; xy[1]=y0;
xy[2]=x0; xy[3]=y0-r;
xy[4]=x0; xy[5]=y0-r;
xy[6]=x0-r; xy[7]=y0;
xy[8]=x0-r; xy[9]=y0;
xy[10]=x0; xy[11]=y0+r;
xy[12]=x0; xy[13]=y0+r;
xy[14]=x0+r; xy[15]=y0;

bx=stangle/45;  //起始八分圆
ex=endangle/45;  //终止八分圆
ben=ex-bx-1;  //扇形所全部占据的八分圆的数量
xy[16]=(double)r*Lcos(stangle);  //起始点的横坐标
xy[17]=(double)r*Lsin(stangle);  //起始点的纵坐标
xy[18]=(double)r*Lcos(endangle);  //终止点的横坐标
xy[19]=(double)r*Lsin(endangle);  //终止点的纵坐标
line(x0+xy[16],y0-xy[17],x0,y0);  //画玄线
line(x0+xy[18],y0-xy[19],x0,y0);  //画玄线
if(bx==1||bx==2||bx==5||bx==6)
    bxd=abs(xy[16]);
```

```
else
    bxd=abs(xy[17]);
if(ex==1||ex==2||ex==5||ex==6)
    exd=abs(xy[18]);
else
    exd=abs(xy[19]);
if(bx==0||bx==2||bx==4||bx==6)
    bxf=0;
else
    bxf=1;
if(ex==0||ex==2||ex==4||ex==6)
    exf=1;
else
    exf=0;

while(x<=xmax) {
    if(tn>=0)
        {
        tn += ( 6 + ((x-y)*4) );
        y--;
        xy[0]--;
        xy[3]++;
        xy[5]++;
        xy[6]++;
        xy[8]++;
        xy[11]--;
        xy[13]--;
        xy[14]--;
        }
    else
        tn += ( (x*4) + 2 );
/*  以下24行，画出扇形所全部占据的八分圆上的点  */
    if(stangle<endangle)
        {
        j=(bx+1)*2;
        for(i=0;i<ben;i++)
            {
            putpixel(xy[j],xy[j+1]);
            j+=2;
            }
        }
    else if(stangle>endangle)
        {
        j=(bx+1)*2;
        for(i=bx+1;i<8;i++)
            {
            putpixel(xy[j],xy[j+1]);
            j+=2;
```

```
            }
        j=0;
        for(i=0;i<ex;i++)
            {
            putpixel(xy[j],xy[j+1]);
            j+=2;
            }
        }
/*  以下6行，画出扇形部分占据的八分圆上的点  */
    i=bx*2;
    if( (x>bxd)^bxf )
        putpixel(xy[i],xy[i+1]);
    i=ex*2;
    if( (x>exd)^exf )
        putpixel(xy[i],xy[i+1]);
    x++;
    xy[1]--;
    xy[2]++;
    xy[4]--;
    xy[7]--;
    xy[9]++;
    xy[10]--;
    xy[12]++;
    xy[15]++;
    }
free(xy);
}
```

9.4 画 椭 圆

9.4.1 算法

利用椭圆的对称性，可以只计算处于第一象限的1/4椭圆周上的点。为了使因变量的变化幅度小于自变量的变化幅度，需要将这1/4的椭圆周分成两段分别进行计算，这两段椭圆弧的分界点为第一象限的椭圆周上斜率为-1的点。

设椭圆中心为(x0,y0)，其处于坐标原点，横轴的长度为2a，纵轴的长度为2b。

设两段椭圆弧的分界点为$M(x_M,y_M)$，根据该点的斜率-1，可得该点坐标值的计算公式如下：

$$x_M=a^2/(a^2+b^2)^{1/2} \tag{9-10}$$

$$y_M=b^2/(a^2+b^2)^{1/2} \tag{9-11}$$

该分界点将第一象限的椭圆弧分为两段。第一段弧上，x的定义域为$[0,x_M]$，y的定义域为$[y_M,b]$，该弧上各点斜率的值域为[-1,0]，因而可直接用与上两节类似的算法对这段弧进行计算处理；第二段弧上，x的定义域为$(x_M,a]$，y的定义域为$[0, y_M)$，该弧上各点斜率的值域为$(-\infty,-1)$，这就需要首先以y=x为对称轴对该弧进行反射变换，然后再进行计算 处

理。

椭圆的扫描转换算法类似与圆，也是通过一个判别变量来确定当x方向增1时，y方向是不变还是减1，下面针对第一段弧介绍椭圆的扫描转换算法。

判别变量的递推公式为：

$$d_1=2b^2+(1-2b)a^2 \tag{9-12}$$

$$d_{i+1}=d_i+(4x_{i-1}+6)b^2 \quad \text{当} d_i<0 \tag{9-13}$$

$$d_{i+1}=(4x_{i-1}+6)b^2+4(1-y_{i-1})a^2 \quad \text{当} d_i\geq 0 \tag{9-14}$$

y_i的取值规则为：若$d_i<0$，则$y_i=y_{i-1}$；若$d_i\geq 0$，则$y_i=y_{i-1}-1$。

具体算法如下：

①令x=0，y=b，画出点(x_0+x,y_0+y)及该点在椭圆周上的其它3个对称点；

②令$dst=2b^2+(1-2b)a^2$;

③若di<0，则：令$dst=dst+(4x+6)b^2$;

　否则：令$dst=dst+(4x+6)b^2+4(1-y)a^2$，令y=y-1；

④令x=x+1，画出点(x_0+x,y_0+y)及该点在椭圆周上的其它3个对称点；

⑤若$x\geq a^2/(a^2+b^2)^{1/2}$，则：结束。

　否则：转到③。

对第二段弧进行反射变换后，其算法与第一段弧的算法完全一样，其具体处理方式见系统程序9-5。

9.4.2　程序

程序中的判别变量及两个半轴的平方都很容易超出2字节整型数的范围，它们必须被定义为4字节的长整型数。

系统程序9-5　VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：画椭圆
输入参数：  x0＝椭圆中心点的横坐标值
            y0＝椭圆中心点的纵坐标值
            r1＝椭圆的横半轴
            r2＝椭圆的纵半轴
返回值：无
*************************************************************************/
void VGABASE::ellipse(int x0,int y0,long r1,long r2)
{
long r；
long r12,r22;  //横半轴及纵半轴的平方
int x,y,xmax;
long tn;  //判别变量
/*  首先计算第一段弧  */
x=0;y=r2;
r12=r1*r1;r22=r2*r2;
xmax=(double)r12/sqrt(r12+r22);
tn=r12-2*r2*r12;
```

```
while(x<=xmax) {
    if(tn<0||y==0)
        tn+=(4*x+2)*r22;
    else
        {
        tn+=(4*x+2)*r22+(1-y)*4*r12;
        y--;
        }
    putpixel(x0+x,y0+y);  //第一象限(屏幕坐标系，下同)
    putpixel(x0-x,y0+y);  //第二象限
    putpixel(x0+x,y0-y);  //第四象限
    putpixel(x0-x,y0-y);  //第三象限
    x++;
    }
/*  以下计算第二段弧  */
r=r1;r1=r2;r2=r;  //反射变换
x=0;y=r2;
r12=r1*r1;r22=r2*r2;
xmax=r12/sqrt(r12+r22);
tn=r12-2*r2*r12;
while(x<=xmax) {
    if(tn<0||y==0)
        tn+=(4*x+2)*r22;
    else
        {
        tn+=(4*x+2)*r22+(1-y)*4*r12;
        y--;
        }
    putpixel(x0+y,y0+x);  //第一象限
    putpixel(x0+y,y0-x);  //第四象限
    putpixel(x0-y,y0+x);  //第二象限
    putpixel(x0-y,y0-x);  //第三象限
    x++;
    }
}
```

9.5 区 域 填 充

9.5.1 概述

前面几节所讨论的图形都属于线条图，在程序中还需要显示另外一类图形——实面积图，实面积图的产生问题即称为区域填充。

区域是指相互连通的一组象素的集合。区域通常由一个封闭的轮廓线来定义，处于一个封闭轮廓线内的所有象素点即构成一个区域。根据轮廓线的定义方式，区域填充问题可分为两类：一是预定义区域填充，在这类填充问题中，区域的边界由一个或一组可解析表示的线条图来定义，根据线条图类型的不同，这类问题又可细分为多种，如多边形填充、圆填充、

椭圆填充等，这类问题也常称为多边形填充，因为在光栅图形系统中，任一个预定义的线条图都可用多边形来完全逼近；二是任意区域填充，这时区域边界是一个已画出的具有一定颜色值的封闭曲线，程序需根据边界的颜色及所给定的一个区域内点通过搜索来确定区域的边界，所给定的这个区域内点称为种子点，所以这类问题也称为种子填充。

根据连通规则、填充方式、区域的几何特征等，还可从其它一些方面对区域填充问题进行分类，区域填充被广泛应用于各种场合，在不同的场合中区域填充的条件和要求都存在着差别，并不存在一种能高效地适用于所有场合的区域填充算法。本书将针对常规的应用介绍一类区域填充算法，并基于该类算法实现如下一些区域填充功能：矩形填充，单个及多个多边形填充，圆、椭圆、扇形填充，无空洞的任意区域填充。所有这些功能都将支持带图案的填充。

与线条图的扫描转换算法相比，区域填充算法要复杂得多，不可能由一个函数来完整地完成某项填充功能，而需要构造一个小的程序系统来对各项填充功能提供支持。下面首先集中介绍各项填充功能的算法，然后介绍程序。

9.5.2　填充原理及算法

因任何一个封闭曲线都可用多边形来表示，因此下面首先针对多边形来讨论区域填充的一般原理及算法，然后讨论各种区域的具体算法。

1．区域内部点的判别准则

对区域进行填充，首先就需要找出处于区域内的所有象素点，当区域的边界给定之后，按如下准则可以判别一个象素点是否处于区域之内：从某一象素点引出一条伸向无穷远的射线，若射线与区域边界的交点数目为奇数，则该点为区域的内点，若射线与区域边界的交点数目为偶数，则该点为区域的外点。

当用上述判别准则进行具体操作时，通常需要给引向无穷远的射线规定一个固定的方向，这里将其确定为向右的水平直线。当固定了射线的方向之后，射线就有可能相交于多边形的某个顶点或重合于多边形的某条边线，这时就需要进行特殊的处理才能得到正确的判别结果。

当向右的水平射线相交于多边形的某个顶点时，若该顶点为Y方向的局部极值点，则，将该点视为两个交点，即认为射线在该顶点上与区域边界相交了两次；否则，按正常情况处理，即认为射线在该顶点上与区域边界相交了一次。

当向右的水平射线重合于多边形的某条水平边线时，若该水平边线的前后两条边线是一上一下的，则将该水平边线的两个端点作为两个交点，即认为射线与该边线相交了两次；若该水平边线的前后两条边线同为向上或同为向下，则只将该水平边线的某一个端点作为交点，即认为射线与边线相交了一次，这时，究竟取那个端点作为交点，在填充效果上存在细微的差别，通常可任取一个。

2．扫描线相关算法

上面仅仅从原理上介绍了区域内点的判别准则，如何利用这一准则来进行区域填充，则还需作进一步的研究。显然不能设想直接用上述准则对显示平面上的每个点逐一判别其内外特性，这样做的计算量是根本无法接受的。

事实上，显示平面上的各个象素点，它们相对于一个给定的区域边界来说，其内外性质

并不是互不关联、随机改变的，也就是说，如果一个象素点处于区域的内部(或外部)，则与之相邻的象素点一般也仍然处于区域的内部(或外部)，只是在特殊的情况下相邻的象素点才具有不同的内外性质，这就是空间相关性。空间相关性自然也体现在每一条扫描线上，即扫描线上相邻的象素点几乎都具有相同的内外性质，只有在遇到扫描线与区域边界的交点时，相邻象素点的内外性质才会不同，这就是所谓的扫描线相关性。

利用扫描线相关性，就用不着对扫描线上的象素点逐一进行测试，而只需求出扫描线与区域边界的交点，这些交点必然将扫描线分成内外交替的段，将扫描线与区域边界的各个交点按其x值的大小进行排序，则交点1与交点2之间的线段必处于区域内，交点2与交点3(如果存在)之间的线段必处于区域外，交点3与交点4之间的线段又处于区域内，依此类推。

根据以上分析可得到扫描线相关算法的基本步骤如下：

①求出区域边界与各条扫描线的全部交点，保存所有交点的x,y值；

②对所保存的全部交点按y值的大小进行排序，对于y值相同的点，则按x的大小排序；

③将排序好的交点依次两两配对，对每对交点之间的扫描线段进行填色。

上面给出的只是一个原则性的算法，其中有四个问题需要在进一步的算法中作出具体处理：(1)如何求得所有交点，这与边界的定义方式及类型有关；(2)如何保存所有交点；(3)如何对交点进行排序；(4)如何进行填色，这与填充图案有关。下面几部分内容将分别讨论这四个问题的解决方法。

3．交点的存储与排序

交点的存储方法和排序方法是联系在一起的，下面介绍三种常用的方法。

(1)y桶排序算法

在求交点之前，对区域所覆盖的每条扫描线准备好一定的存储空间，用来存放该扫描线的y值，及若干个交点的x值，该存储空间称为y桶。每条扫描线按其y值的大小依次对应于一个y桶，y桶本身已包含了对各交点y值的排序。所求出的每个交点按其y值存入相应的y桶中，这时只需对每个y桶中的各个交点按它们的x值进行排序，排序可以在存放交点时进行，也可在求出全部交点后再进行。

这种方法具有很快的排序速度，在各扫描线上的交点数相同时，这种方法所占用的存储空间也较小。但在使用这种方法时，必须要能事先确定出区域所覆盖的扫描线及每条扫描线上所可能具有的最大交点数，因此，这种方法不适合于任意区域填充及多边形填充，因为多边形能够产生很复杂的凹入和空洞情况，其主要适用于圆填充、椭圆填充、扇形填充。

(2)链表排序算法

用一个双向链表来存储所有的交点，链表中的每个记录保存一个交点，每求得一个交点就按排序要求将该点插入到链表中的某个位置上，即在保存交点的同时完成排序。总是从上一个交点的插入位置开始来搜索当前交点的插入位置，通常各个交点都是按顺序求得的，因此往往只需很少的搜索步骤就能得到交点的插入位置，而且在链表上插入交点时，只需改写前后两个记录的指针，而无需实际移动数据，所以这种方法通常都具有很快的排序速度。由于这种方法不需要事先确定扫描线的数量，也不需要限制每条扫描线上的交点数量，因此这种方法适合于任意区域填充和多边形填充。

链表中的每个记录必须包含有链域，而链域所占的空间往往比交点本身所占的空间还要大，因此这种方法需要占用较大的存储空间。此外，在不是顺序求得各交点，而是跳跃获得

各交点时，如用对称方法求圆的交点，这种方法的排序速度就会很慢。

(3)边相关算法

这是专门用于多边形填充的一种算法。在多边形填充中，相邻扫描线上的交点是与多边形的边线相关的，即当一条扫描线与某一边线有交点时，那么相邻的扫描线一般也与该边线有交点，除非该扫描线超出了该边线的y值范围。利用边的相关性，就没有必要事先算出每条边线与各扫描线的全部交点，而只需以边线为单位，对每条边建立一个边记录，记下每条边的斜率及两个端点的x，y值。然后从上到下或从下到上，依次对与当前扫描线相交的边进行扫描转换，边转换边填充。

与链表排序算法相比，这种方法能大大节省存储空间，在速度上则基本相当。与上面两种算法相比，这种算法的实现要复杂一些。

4．圆、椭圆、扇形的填充算法

对以圆、椭圆或扇形为边界所定义的区域，可以预先确定区域所覆盖的扫描线，且圆和椭圆与每条扫描线的交点只会有两个，扇形与扫描线的交点最多只能达到4个，因此对这类填充问题采用了y桶排序算法来保存和排序交点。

采用9.3节、9.4节中所介绍的扫描转换算法来求得区域边界与各扫描线的交点，在求交点时需要注意对边界中水平直线的处理，经过扫描转换，圆、椭圆及扇形上的多个相邻点往往会处于同一扫描线上，这时就只能从处于同一扫描线上的多个相邻点中取一个或两个作为交点，而不能将它们全部作为交点，对于扇形则还需要对极值点进行处理。

下面具体给出圆的填充算法，

设圆心的坐标为(x0,y0)，圆的半径为R。

①定义2R+1个y桶，每个y桶存储两个交点，第一个y桶对应的y值为y0-R，其余逐步递增；

②令x=0，y=R；令dst=3-2R;

③若di<0，则：令dst=dst+4x+2;

　否则：将点(x0+x,y0+y)、(x0-x,y0+y)、(x0+x,y0-y)、(x0-x,y0-y)存入y桶；

　　令dst=dst+4(x-y)+6；令y=y-1；

④若x>0，则：将点(x0-y,y0-x)、(x0+y,y0-x)存入y桶；

　将点(x0-y,y0+x)、(x0+y,y0+x)存入y桶；

　令x=x+1；

⑤若x≥R×sin45，则：转到⑥；

　否则：转到③；

⑥对每个y桶内两个交点间的扫描线进行填色。

椭圆和扇形的填充算法与圆的填充算法类似，这里不再给出，具体可参见系统程序9-8中的两个函数VGABASE::sectorfill()和VGABASE::ellipsefill()。

5．多边形及任意区域填充算法

多边形填充和任意区域填充都采用了链表排序算法。

在多边形填充中，每条边线与各扫描线的交点采用9.2节中所介绍的扫描转换算法求得，在求交点时需要对处于同一扫描线上的多个相邻点及极值点进行处理。程序中提供了对多个多边形同时进行填充的功能，利用这一功能可实现对带空洞区域的填充。 多边形填充

的具体算法参见系统程序9-7中的函数LFILL::poly()。

在任意区域填充中，区域边界是已画出的一个封闭围线，通过给定该围线的颜色值及区域内的一个种子点来确定所要填充的区域，程序首先通过搜索得到区域边界上的每个点，然后确定出边界与各扫描线的交点并将交点存入交点链表中，最后根据交点链表完成填充。搜索边界点的步骤如下：

①从种子点出发，沿水平方向向右搜索，直到得到边界上的某个点，将该点置为当前点，并记该点为第一个边界点；

②保存当前点；

③依次读与当前点连通的8个点，若所读出的某个点为边界点，则以填充色(不同于边界色)写该点，若该点为第一个边界点，则结束，否则，将该点置为当前点，转到②；

若所读出的8个点都不为边界点，则转到④；

④若当前点不为第一个边界点，则将上一个点置为当前点，转到②；若当前点为第一个边界点则表明边界不封闭，结束。

其中步骤④主要用于处理一个边界点与多个边界点连通的情况。任意区域填充的具体算法参见系统程序9-8中的函数VGABASE::fillarea()。

6．填充图案

在很多情况下可能不希望以一种颜色填满整个区域，而需要以一定的图案如斜线、网格等来对区域进行填充，程序实现了对图案填充的支持。程序对各类区域的填充都采用了扫描线相关算法，在这类算法中，填充的图案对交点的获取、存储及排序是没有影响的，因此图案填充在程序中可作为一个独立的问题来处理。

在程序中用8个字节来定义一种填充图案。该8个字节对应于一个8×8点阵的图象掩模，当某个字节中的某位为1时，该位所对应的象素点在填充时就将写上填充色，为0时则保持原色不变，从屏幕左上角开始，向右向下每隔8个象素点重复使用该图象掩模即可实现对整个屏幕填充图案的定义。程序中预定义了10种填充图案，并留有一个自定义填充图案的接口，各个预定义填充图案的形状及掩码见系统程序9-7。填充图案亦称为填充模式。

在图案填充中，对那些不需填色的象素点可采用两种处理方式，一是将它们的颜色置为0，二是保留它们的颜色不变。第二种方式具有更大的灵活性，特别在16色以上的色彩模式中往往都需要以这种方式进行图案填充，但使用这种方式时，就只能用写点操作而不能用写扫描线操作来进行填充，因此其速度较慢，程序中采用了第二种方式。

9.5.3 区域填充基础程序

各种类型区域填充都使用了类似的算法，并采用了相同的图案填充处理方式，因此首先实现了一组基础程序来提供对y桶排序算法、链表排序算法及图案填充的支持。y桶排序算法及链表排序算法的有关数据及功能分别包含在类YLIFF和类LFILL中，由于这两种算法具有一些共同的地方，因此这两个类派生于同一个基类FILL。系统程序9-6给出了这组基础填充程序的说明，系统程序9-7给出了这组程序的定义，这组程序主要供其它系统程序使用，在应用程序中一般不需直接调用该组程序。

系统程序9-6　FILLBASE.H

```
--------------------------------------------------------------------------
/*  扫描线相关算法类  */
class FILL {
public:
    static void setstyle(int fstyle);  //选择填充图案
    static int getstyle(void);  //取当前的填充图案
    static void setpattern(unsigned char *p);  //设置自定义填充图案
    static void scanlinestyle(int x1,int x2,int y);  //带图案画扫描线
    void line(int x1,int y1,int x2,int y2);
      //将一条直线的所有交点加入y桶或交点链表中
    virtual void inspole(int x,int y) { }
      //将一个多边形的所有交点加入y桶或交点链表中
    static int dataptr;  //当前填充图案的数据指针
protected:
    static unsigned char STD[96];  //各种填充图案的掩码
    static int STDN;  //填充图案的数量
    static VGABASE *vga;  //当前显示模式类对象的指针

    FILL(VGABASE *v) { vga=v; }  //构造函数，获取当前的显示模式
    } ;

/*  交点链表数据结构  */
struct LFS {
    LFS *next,*prev;  //链表中下一个和上一个记录的指针
    int x,y;  //交点的x，y值

    LFS(int x0,int y0) { x=x0; y=y0; }
    void insbefore(LFS *lfs);  //插入一个交点
    void del();  //删除一个交点
/*  以下重载了四个关系操作符，它们主要在搜索插入点时使用  */
    int operator>(LFS lp) {
        if(y>lp.y || (y==lp.y&&x>lp.x) ) return(1);
        else return(0); }
    int operator<(LFS lp) {
        if( y<lp.y || (y==lp.y&&x<lp.x) ) return(1);
        else return(0); }
    int operator>=(LFS lp) {
        if(y>lp.y || (y==lp.y&&x>=lp.x) ) return(1);
        else return(0); }
    int operator<=(LFS lp) {
        if( y<lp.y || (y==lp.y&&x<=lp.x) ) return(1);
        else return(0); }
    } ;

/*  链表排序算法类  */
class LFILL : public FILL
    {
```

```
    LFS *beg,*end;  //链表起始记录和终止记录的指针
    LFS *cur;  //最近一次所插入记录的指针
public:
    LFILL(VGABASE *v);
    ~LFILL();
    void inspole(int x,int y);  //插入一个点
    void inspole(LFS *lp);  //插入一个点
    void add(LFILL *lfp);  //将两个链表相加
    int ifincl(LFILL *lfp);  //判别两个链表所对应的区域是否完全包含
    void poly(int n,int *border);  //将一个多边形的所有交点加入链表中
    void draw();  //对链表所定义的区域进行填充
    void save(void);  //将所有交点存盘。调试用
    } ;

/*  y桶数据结构  */
struct YFS {
    int n;  //一条扫描线上的交点数量
    int x[4];  //各交点的x值
    } ;

/*  y桶排序算法类  */
class YFILL : public FILL
    {
    YFS *yfs;
    int Y1,Y2;  //所有y桶中的最小、最大y值
public:
    YFILL(VGABASE *v,int y1,int y2);
   ~YFILL();
    void inspole(int x,int y);  //插入一个交点
    void supple(void);  //弥补遗漏的交点
    void draw();  //对y桶所定义的区域进行填充
    void save(void);  //将所有交点存盘。调试用
    } ;
```

系统程序9-7 FILLBASE.CPP

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <math.h>

#include "\vga\vga\vga.h"
#include "\vga\xms\xms.h"
#include "\vga\vga\fillbase.h"

VGABASE *FILL::vga;
int FILL::dataptr=0;
int FILL::STDN=12;
```

```
unsigned char FILL::STD[96]={  //12种填充图案的掩码
    0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,  //全填充
    0xff,0xff,0x00,0x00,0xff,0xff,0x00,0x00,  //细水平线
    0xff,0xff,0xff,0xff,0x00,0x00,0x00,0x00,  //粗水平线
    0x03,0x06,0x0c,0x18,0x30,0x60,0xc0,0x81,  //细左斜线
    0x1e,0x3c,0x78,0xf0,0xe1,0xc3,0x87,0x0f,  //粗左斜线
    0xc0,0x60,0x30,0x18,0x0c,0x06,0x03,0x81,  //细右斜线
    0x78,0x3c,0x1e,0x0f,0x87,0xc3,0xe1,0xf0,  //粗右斜线
    0x41,0x22,0x14,0x08,0x14,0x22,0x41,0x80,  //斜网格
    0x44,0xff,0x44,0x44,0x44,0xff,0x44,0x44,  //方网格
    0x55,0xaa,0x55,0xaa,0x55,0xaa,0x55,0xaa,  //细点
    0x30,0x30,0x03,0x03,0x30,0x30,0x03,0x03,  //粗点
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  //空。用于用户自定义图案

/*  用于带图案填充的画扫描线操作。用写点函数实现  */
void FILL::scanlinestyle(int x1,int x2,int y)
{
unsigned char c,c1,b;
int x;

if(x1>x2)
    {
    x=x1; x1=x2; x2=x;
    }
c=c1=STD[dataptr+(y&7)];
b=x1&7;
c<<=b;
for(x=x1;x<=x2;x++)
    {
    if( c&0x80 )
        vga->putpixel(x,y);
    b++;
    if(b==8)
        { b=0; c=c1; }
    else
        c<<=1;
    }
}

/*  设置填充模式  */
void FILL::setstyle(int fst)
{
if(fst>=0&&fst<STDN)
    dataptr=fst*8;  //将数据指针指向相应的掩码
}

/*  获取填充图案  */
int FILL::getstyle(void)
```

```
{
return(dataptr/8);
}

/*  设置用户自定义填充模式  */
void FILL::setpattern(unsigned char *s)
{
dataptr=(STDN-1)*8;
memcpy(STD+dataptr,s,8);
}

/*  在y桶或边点链表中写一条直线。不写两个端点，且只写y值不同的点  */
void FILL::line(int x1,int y1,int x2,int y2)
{
int i,p,n,x,y,tn,my;

if( abs(y1-y2)<=1 )  //水平或近水平直线
    return;
if( x1==x2 )  //垂直直线
    {
    if(y1>y2)
        { y=y2; y2=y1; y1=y; }
    for(i=y1+1;i<y2;i++)
        inspole(x1,i);
    return;
    }
/*  以下采用Bresenham算法生成直线中的各点，但只取那些y值有差异的点  */
if( abs(y2-y1) <= abs(x2-x1) )
    {
    if( (y2<y1&&x2<x1) || (y1<=y2&&x1>x2) )
        { x=x2; y=y2; x2=x1; y2=y1; x1=x; y1=y; }
    if( y2>=y1 && x2>=x1 )
        {
        x=x2-x1; y=y2-y1;
        p=2*y; n=2*x-2*y; tn=x;
        my=y2;
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else
                {
                tn+=n; y1++;
                if(y1<my)
                    inspole(x1,y1);
                }
            x1++;
            }
        }
    else
```

```
        {
        x=x2-x1; y=y2-y1;
        p=-2*y; n=2*x+2*y; tn=x;
        my=y2;
        while(x1<=x2) {
            if(tn>=0) tn-=p;
            else
                {
                tn+=n; y1--;
                if(y1>my)
                    inspole(x1,y1);
                }
            x1++;
            }
        }
    }
else
    {
    x=x1; x1=y2; y2=x; y=y1; y1=x2; x2=y;
    if( (y2<y1&&x2<x1) || (y1<=y2&&x1>x2) )
        { x=x2; y=y2; x2=x1; y2=y1; x1=x; y1=y; }
    if( y2>=y1 && x2>=x1 )
        {
        x=x2-x1; y=y2-y1;
        p=2*y; n=2*x-2*y; tn=x;
        x1++;
        while(x1<x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1++; }
            inspole(y1,x1);
            x1++;
            }
        }
    else
        {
        x=x2-x1; y=y2-y1;
        p=-2*y; n=2*x+2*y; tn=x;
        x1++;
        while(x1<x2) {
            if(tn>=0) tn-=p;
            else { tn+=n; y1--; }
            inspole(y1,x1);
            x1++;
            }
        }
    }
}
```

```
/*  在交点链表中前插一个交点  */
void LFS::insbefore(LFS *lfs)
{
next=lfs;
prev=lfs->prev;
prev->next=this;
lfs->prev=this;
}

/*  在交点链表中删除一个交点  */
void LFS::del()
{
if(prev!=0&&next!=0)
    {
    prev->next=next;
    next->prev=prev;
    }
}

/*  构造函数。生成交点链表中的第一个和最后一个交点  */
LFILL::LFILL(VGABASE *v) : FILL(v)
{
beg=new LFS(-1,-1);
end=new LFS(8888,8888);
beg->prev=end->next=0;
beg->next=end;
end->prev=beg;
cur=end;
}

/*  析构函数。释放空间  */
LFILL::~LFILL()
{
LFS *p,*p1;
p=beg;
while(p!=0) {
    p1=p->next;
    delete p;
    p=p1;
    }
}

/*  在交点链表中写一个点  */
void LFILL::inspole(int x,int y)
{
LFS *p;
p=cur;
cur=new LFS(x,y);
```

```
if( *cur>*p )
    {
    p=p->next;
    while(1) {
        if( *cur>*p )
            p=p->next;
        else
            { cur->insbefore(p); breaK; }
        }
    }
else
    {
    p=p->prev;
    while(1) {
        if( *cur<*p )
            p=p->prev;
        else
            {
            cur->insbefore(p->next);
            breaK;
            }
        }
    }
}

/*  将另一个交点链表中的点写到本链表中  */
void LFILL::inspole(LFS *lp)
{
LFS *p;
p=cur;
cur=lp;

if( *lp>*p )
    {
    p=p->next;
    while(1) {
        if( *lp>*p )
            p=p->next;
        else
            { lp->insbefore(p); breaK; }
        }
    }
else
    {
    p=p->prev;
    while(1) {
        if( *lp<*p )
```

```
            p=p->prev;
        else
            { lp->insbefore(p->next); breaK; }
        }
    }
}

/*  将两个交点链表合并  */
void LFILL::add(LFILL *lfp)
{
LFS *begp=lfp->beg;
LFS *lp=lfp->beg->next;
while(lp->next!=0) {
    lp->del();
    inspole(lp);
    lp=begp->next;
    }
}

/*  在交点链表中写一个多边形  */
void LFILL::poly(int n,int *border)
{
int i,j,nn;
int *x,*y;

x=(int *)calloc(n+3,2);
y=(int *)calloc(n+3,2);
j=0;
for(i=1;i<=n;i++)
    {
    x[i]=border[j]; j++;
    y[i]=border[j]; j++;
    if( i>1 && x[i]==x[i-1] && y[i]==y[i-1] )
        { i--; n--; }
    }
for(i=2;i<n;i++)
    {
    if( y[i]==y[i-1] && y[i]==y[i+1] )
        {
        memmove(&x[i],&x[i+1],(n-i)*2);
        memmove(&y[i],&y[i+1],(n-i)*2);
        n--; i--;
        }
    }
if( y[1]==y[2] && y[1]==y[n] )
    {
    memmove(&x[1],&x[2],(n-1)*2);
    memmove(&y[1],&y[2],(n-1)*2);
```

```
    n--;
    }
if( y[n]==y[n-1] && y[n]==y[1] )
    n--;
x[0]=x[n]; y[0]=y[n];
x[n+1]=x[1]; y[n+1]=y[1];
x[n+2]=x[2]; y[n+2]=y[2];

for(i=1;i<=n;i++)
    {
    j=i+1;
    if(y[i]==y[j])
        {
        if( (y[i]<=y[i-1] && y[j]<=y[j+1]) || (y[i]>=y[i-1] && y[j]>=y[j+1]) )
            {
            inspole(x[i],y[i]);
            inspole(x[j],y[j]);
            }
        else
            inspole(x[i],y[i]);
        }
    else if(y[i]!=y[i-1])
        {
        if( (y[i]<y[i-1] && y[i]<y[i+1]) || (y[i]>y[i-1] && y[i]>y[i+1]) )
            {
            inspole(x[i],y[i]);
            inspole(x[i],y[i]);
            }
        else
            inspole(x[i],y[i]);
        }
    }
for(i=1;i<=n;i++)
    line(x[i],y[i],x[i+1],y[i+1]);
free(x); free(y);
}

/*  将交点链表中的实行画到屏幕上  */
void LFILL::draw(void)
{
LFS *p1,*p2;
p1=beg->next;
if(dataptr==0)
    {
    while(p1!=0) {
        p2=p1->next;
        if(p2==0)
            breaK;
```

```
        vga->scanline(p1->x,p2->x,p1->y);
        p1=p2->next;
        }
    }
else
    {
    while(p1!=0) {
        p2=p1->next;
        if(p2==0)
            breaK;
        scanlinestyle(p1->x,p2->x,p1->y);
        p1=p2->next;
        }
    }
}

/*  保存交点链表中的数据。调试用  */
void LFILL::save(void)
{
LFS *p1,*p2;
FILE *fp;

fp=fopen("LFILL.DAT","w");
fprintf(fp,"%d  %d\n\n",beg->x,beg->y);
p1=beg->next;
while(p1!=0) {
    p2=p1->next;
    fprintf(fp,"%d  %d\n",p1->x,p1->y);
    fprintf(fp,"%d  %d\n\n",p2->x,p2->y);
    p1=p2->next;
    if(p2==0)
        breaK;
    }
fclose(fp);
}

/*  y桶排序算法类构造函数。确定y桶大小，分配y桶空间  */
YFILL::YFILL(VGABASE *v,int y1,int y2) : FILL(v)
{
if(y2<y1)
    { Y1=y2; Y2=y1; }
else
    { Y1=y1; Y2=y2; }
yfs=(YFS *)calloc(Y2-Y1+1,sizeof(YFS));
}

/*  析构函数。释放y桶空间  */
YFILL::~YFILL()
```

```
{
free(yfs);
}

/*  在y桶中写一个点  */
void YFILL::inspole(int x,int y)
{
if(y>=Y1&&y<=Y2)
    {
    int yy=y-Y1;
    yfs[yy].x[yfs[yy].n]=x;
    yfs[yy].n++;
    }
}

/*  补上y桶中的空行。在圆和椭圆的填充算法中，1/8圆弧的交结处有可能漏掉一行，因而需要该函
数  */
void YFILL::supple(void)
{
int i,n;
n=Y2-Y1+1;
for(i=1;i<n;i++)
    if(yfs[i].n<=1&&yfs[i-1].n>=2)
        {
        yfs[i].n=2;
        yfs[i].x[0]=yfs[i-1].x[0];
        yfs[i].x[1]=yfs[i-1].x[1];
        }
}

/*  将y桶中的图形输出到屏幕上  */
void YFILL::draw(void)
{
int i,x,n,y;
int *xx;

n=Y2-Y1+1;
y=Y1;
for(i=0;i<n;i++)
    {
    xx=yfs[i].x;
    if(yfs[i].n==4)
        {
        if(xx[0]>xx[1])
            {
            x=xx[1];
            xx[1]=xx[0];
            xx[0]=x;
```

```
            }
        if(xx[2]>xx[3])
            {
            x=xx[3];
            xx[3]=xx[2];
            xx[2]=x;
            }
        if( xx[2]<xx[1] && xx[3]>xx[0] )
            {
            x=xx[2];
            xx[2]=xx[1];
            xx[1]=x;
            }
        if(dataptr==0)
            {
            vga->scanline(xx[2],xx[3],y);
            vga->scanline(xx[0],xx[1],y);
            }
        else
            {
            scanlinestyle(xx[2],xx[3],y);
            scanlinestyle(xx[0],xx[1],y);
            }
        }
    else if(yfs[i].n==2)
        {
        if(dataptr==0)
            vga->scanline(xx[0],xx[1],y);
        else
            scanlinestyle(xx[0],xx[1],y);
        }
    y++;
    }
}

/*  保存y桶中的数据。调试用  */
void YFILL::save(void)
{
int i,n;
FILE *fp;

fp=fopen("YFILL.DAT","w");
n=Y2-Y1+1;
for(i=0;i<n;i++)
    fprintf(fp,"%3d  %d   %d  %d  %d  %d\n",Y1+i,yfs[i].n,yfs[i].x[0],
    yfs[i].x[1],yfs[i].x[2],yfs[i].x[3]);
fclose(fp);
}
```

9.5.4　区域填充程序

这里给出VGABASE类各个区域填充函数的定义，这些函数都要用到上面所介绍的区域填充基础程序。

系统程序9-7　VGAFILL.CPP

```
--------------------------------------------------------------------------
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <math.h>

#include "\vga\vga\vga.h"
#include "\vga\xms\xms.h"
#include "\vga\vga\fillbase.h"

static double SIN45=0.707106781186548;

/*************************************************************************
功能：设置填充模式
输入参数：fst＝填充模式的编号，其具体含义如下：
                0  全填充，此为默认值
                1  细水平线
                2  粗水平线
                3  细左斜线
                4  粗左斜线
                5  细右斜线
                6  粗右斜线
                7  斜网格
                8  方网格
                9  细点
                10  粗点
                11  用户自定义图案
返回值：无
*************************************************************************/
void VGABASE::setfillstyle(int fst)
{
FILL::setstyle(fst);
}

/*************************************************************************
功能：取当前填充模式的编号
输入参数：无
返回值：int＝当前填充模式的编号，其具体含义同前。
*************************************************************************/


int VGABASE::getfillstyle(void)
{
```

```
return FILL::getstyle();
}

/*************************************************************************
功能：设置用户自定义填充图案
输入参数：s＝8字节填充掩码的指针
返回值：无
    调用该函数后，填充图案的类型将自动设为11，即用户自定义类型。
*************************************************************************/
void VGABASE::setfillpattern(unsigned char *s)
{
FILL::setpattern(s);
}

void VGABASE::bar(int x1,int y1,int x2,int y2)
{
int i;
if(FILL::dataptr==0)
/*  如果填充模式为0，即单色填充，则直接调用画扫描线操作完成，这能获得比图案填充快得多的速
度  */
    {
    for(i=y1;i<=y2;i++)
        scanline(x1,x2,i);
    }
else
/*  带图案的填充  */
    {
    for(i=y1;i<=y2;i++)
        FILL::scanlinestyle(x1,x2,i);
    rectangle(x1,y1,x2,y2);
    }
}

/*************************************************************************
功能：填充一个多边形
输入参数：n＝多边形的顶点数
          border＝多边形各顶点的坐标
返回值：无
    输入参数的格式同VGABASE::poly(int,int *)，具体可参见示例程序9-1和示例程序9-2。
*************************************************************************/
void VGABASE::polyfill(int n,int *border)
{
LFILL lfill(this);
lfill.poly(n,border);
lfill.draw();
}
/*************************************************************************
功能：填充多个多边形
```

```
输入参数：border＝各多边形的顶点数及各多边形各顶点的坐标
返回值：无
    输入参数的格式同VGABASE::poly(int *)，具体可参见示例程序9-1和示例程序9-2。
*************************************************************************/
void VGABASE::polyfill(int *border)
{
int n,i=0;
LFILL lf(this);
while(1) {
    n=border[i];
    if(n<=0)
        breaK;
    i++;
    lf.poly(n,border+i);
    i += (n*2);
    }
lf.draw();
}

/*************************************************************************
功能：填充圆
输入参数：  x0＝圆心的横坐标
            y0＝圆心的纵坐标
            r＝半径
*************************************************************************/
void VGABASE::circlefill(int x0,int y0,int r)
{
int i;
long tn,x,y;
long xmax;
YFILL yfill(this,y0-r,y0+r);

y=r; x=0;
xmax=(double)r*SIN45;
tn=(1-r*2);
while(x<=xmax) {
    if(tn>=0)
        {
        tn += ( 6 + ((x-y)<<2) );
        yfill.inspole(x0+x,y0+y); //45度到90度(屏幕坐标系，下同)
        yfill.inspole(x0-x,y0+y); //90度到135度
        yfill.inspole(x0-x,y0-y); //225度到270度
        yfill.inspole(x0+x,y0-y); //270度到315度
        y--;
        }
    else
        tn += ( (x<<2) + 2 );
    yfill.inspole(x0+y,y0+x); //0度到45度
```

```
    yfill.inspole(x0-y,y0+x); //135度到180度
    if(x>0)  //避免正中的水平直线重复计算交点
        {
        yfill.inspole(x0-y,y0-x); //180度到225度
        yfill.inspole(x0+y,y0-x); //315度到360度
        }
    x++;
    }
yfill.supple();
yfill.draw();
}

/*************************************************************************
功能：填充扇形
输入参数：  x0＝圆心的横坐标
            y0＝圆心的纵坐标
            r＝半径
            stangle＝扇形的起始角度(0～360)
            endangle＝扇形的终止角度(0～360)
返回值：无
*************************************************************************/
void VGABASE::sectorfill(int x0,int y0,int r,int stangle,int endangle)
{
int i,j;
long tn,x,y;
long xmax;
int *xy;
int bx,ex,bxd,exd,bxf,exf,ly1,ly2;

bx=(stangle-1)/45;
if(bx<0) bx=7;
ex=endangle/45;
xy=(int *)calloc(4,2);  //存放扇形圆弧的起始和终止点的坐标
xy[0]=x0+(double)r*Lcos(stangle);
xy[1]=y0-(double)r*Lsin(stangle);
xy[2]=x0+(double)r*Lcos(endangle);
xy[3]=y0-(double)r*Lsin(endangle);

bxd=abs(y0-xy[1]);
exd=abs(y0-xy[3]);

if(bx==0||bx==1||bx==4||bx==5)
    bxf=0;
else
    bxf=1;
if(ex==0||ex==1||ex==4||ex==5)
    exf=1;
else
```

```
    exf=0;
/*  以下9行，计算扇形所覆盖扫描线的最小y值，放入变量ly1  */
if( (endangle>=90 && (stangle<=90||stangle>endangle)) ||
(endangle<90 && (stangle>endangle&&stangle<=90)) )
    ly1=y0-r;
else
    {
    ly1=xy[1];
    if(xy[3]<ly1)
        ly1=xy[3];
    }
/*  以下9行，计算扇形所覆盖扫描线的最大y值，放入变量ly2  */
if( (endangle>=270&&(stangle<=270||stangle>endangle)) ||
(endangle<270&&(stangle>endangle&&stangle<=270)) )
    ly2=y0+r;
else
    {
    ly2=xy[3];
    if(xy[1]>ly2)
        ly2=xy[1];
    }
YFILL yfill(this,ly1,ly2);

/*  以下21行，处理极值点  */
if( (xy[1]<=y0&&xy[0]>=x0) || (xy[1]>=y0&&xy[0]<=x0) )
    yfill.inspole(xy[0],xy[1]);
else
    {
    yfill.inspole(xy[0],xy[1]);
    yfill.inspole(xy[0],xy[1]);
    }
if( (xy[3]<=y0&&xy[2]<=x0) || (xy[3]>=y0&&xy[2]>=x0) )
    yfill.inspole(xy[2],xy[3]);
else
    {
    yfill.inspole(xy[2],xy[3]);
    yfill.inspole(xy[2],xy[3]);
    }
if( (xy[1]>y0&&xy[3]>y0) || (xy[1]<y0&&xy[3]<y0) )
    {
    yfill.inspole(x0,y0);
    yfill.inspole(x0,y0);
    }
else if( (xy[1]>=y0&&xy[3]<y0) || (xy[1]<y0&&xy[3]>=y0) )
    yfill.inspole(x0,y0);
/*  以下4行，处理两条玄线  */
if(y0!=xy[1])  //不为水平线
    yfill.line(x0,y0,xy[0],xy[1]);
```

```
if(y0!=xy[3])  //不为水平线
    yfill.line(xy[2],xy[3],x0,y0);

y=r; x=0;
xmax=(double)r*SIN45;
tn=(1-r*2);
while(x<=xmax) {
    if(tn>=0)
        {
        tn += ( 6 + ((x-y)<<2) );
        if( (ex>bx&&bx<6&&ex>6) || (endangle<stangle&&(bx<6||ex>6)) ||
        (bx==6&&((y>bxd)^bxf)&&y!=bxd) || (ex==6&&((y>exd)^exf)&&y!=exd) )
            yfill.inspole(x0+x,y0+y);  //45度到90度
        if( (ex>bx&&bx<5&&ex>5) || (endangle<stangle&&(bx<5||ex>5)) ||
        (bx==5&&((y>bxd)^bxf)&&y!=bxd) || (ex==5&&((y>exd)^exf)&&y!=exd) )
            yfill.inspole(x0-x,y0+y);  //90度到135度
        if( (ex>bx&&bx<2&&ex>2) || (endangle<stangle&&(bx<2||ex>2)) ||
        (bx==2&&((y>bxd)^bxf)&&y!=bxd) || (ex==2&&((y>exd)^exf)&&y!=exd) )
            yfill.inspole(x0-x,y0-y);  //225度到270度
        if( (ex>bx&&bx<1&&ex>1) || (endangle<stangle&&(bx<1||ex>1)) ||
        (bx==1&&((y>bxd)^bxf)&&y!=bxd) || (ex==1&&((y>exd)^exf)&&y!=exd) )
            yfill.inspole(x0+x,y0-y);  //270度到315度
        y--;
        }
    else
        tn += ( (x<<2) + 2 );
    if( (ex>bx&&bx<7&&ex>7) || (endangle<stangle&&(bx<7||ex>7)) ||
    (bx==7&&((x>bxd)^bxf)&&x!=bxd) || (ex==7&&((x>exd)^exf)&&x!=exd) )
        yfill.inspole(x0+y,y0+x);  //0度到45度
    if( (ex>bx&&bx<4&&ex>4) || (endangle<stangle&&(bx<4||ex>4)) ||
    (bx==4&&((x>bxd)^bxf)&&x!=bxd) || (ex==4&&((x>exd)^exf)&&x!=exd) )
        yfill.inspole(x0-y,y0+x);  //135度到180度
    if(x>0)
        {
        if( (ex>bx&&bx<3&&ex>3) || (endangle<stangle&&(bx<3||ex>3)) ||
        (bx==3&&((x>bxd)^bxf)&&x!=bxd) || (ex==3&&((x>exd)^exf)&&x!=exd) )
            yfill.inspole(x0-y,y0-x);  //180度到225度
        if( (ex>bx&&bx<0&&ex>0) || (endangle<stangle&&(bx<0||ex>0)) ||
        (bx==0&&((x>bxd)^bxf)&&x!=bxd) || (ex==0&&((x>exd)^exf)&&x!=exd) )
            yfill.inspole(x0+y,y0-x);  //315度到360度
        }
    x++;
    }
yfill.supple();
yfill.draw();
free(xy);
}
```

```
/*************************************************************************
功能：填充椭圆
输入参数：  x0＝椭圆中心点的横坐标值
            y0＝椭圆中心点的纵坐标值
            r1＝椭圆的横半轴
            r2＝椭圆的纵半轴
返回值：无
*************************************************************************/
void VGABASE::ellipsefill(int x0,int y0,long r1,long r2)
{
long r,xy,r12,r22;
int x,y,xmax,ymax;
long tn,ir12,ir22;
YFILL yfill(this,y0-r2,y0+r2);

x=0;y=r2;
r12=r1*r1;r22=r2*r2;ir12=r12;ir22=r22;
xmax=(double)r12/sqrt(r12+r22);
tn=r12-2*r2*r12;
while(x<=xmax) {
    if(tn<0||y==0)
        tn+=(4*x+2)*ir22;
    else
        {
        tn+=(4*x+2)*ir22+(1-y)*4*ir12;
        yfill.inspole(x0+x,y0+y);
        yfill.inspole(x0-x,y0+y);
        yfill.inspole(x0+x,y0-y);
        yfill.inspole(x0-x,y0-y);
        y--;
        }
    x++;
    }

r=r1;r1=r2;r2=r;
x=0;y=r2;
r12=r1*r1;r22=r2*r2;ir12=r12;ir22=r22;
xmax=(int)( (double)r12/sqrt(r12+r22)-0.1);
tn=r12-2*r2*r12;
while(x<=xmax) {
    if(tn<0||y==0)
        tn+=(4*x+2)*ir22;
    else
        {
        tn+=(4*x+2)*ir22+(1-y)*4*ir12;
        y--;
        }
    yfill.inspole(x0+y,y0+x);
```

```
    yfill.inspole(x0-y,y0+x);
    if(x>0)
        {
        yfill.inspole(x0+y,y0-x);
        yfill.inspole(x0-y,y0-x);
        }
    x++;
    }
yfill.supple();
yfill.draw();
}

/*************************************************************************
功能：任意区域填充
输入参数：  x0＝种子点的横坐标
            y0＝种子点的纵坐标
            bcolor＝边界颜色值
返回值：无
*************************************************************************/
void VGABASE::fillarea(int x0,int y0,union COLOR bcolor)
{
int i,j,K,f;
static int fxy[8][2]={{1,0},{1,-1},{0,-1},{-1,-1},{-1,0},{-1,1},{0,1},{1,1}};
//  8个连通点的增亮
int *xx,*yy,x,y;
int cK,pK;
int n=0；
int maxxyn=4096;  //所允许的边界点的最大数量
LFILL lfill(this);

xx=(int *)calloc(maxxyn,2);  //存放所有边界点的x坐标值
yy=(int *)calloc(maxxyn,2);  //存放所有边界点的y坐标值

/*  以下循环，从种子点出发，向右搜索得到第一个边界点  */
for(i=x0;i<WIDE;i++)
    if( getpixel(i,y0).dword==bcolor.dword )
        {
        xx[0]=i;
        yy[0]=y0;
        n++;
        breaK;
        }
if(n==0)  //未找到边界点
    goto RETU;  //非正常终止

f=0;
K=0;
/*  以下循环搜索得到所有的边界点，放入xx[]，yy[]  */
```

```
while(1) {
    j=f;
    for(i=0;i<8;i++)
        {
        if( i==0||i==2||i==4||i==6 )
            j-=i;
        else
            j+=i;
        if(j<0) j+=8;
        if(j>=8) j-=8;
        x=xx[K]+fxy[j][0];
        y=yy[K]+fxy[j][1];
        if( getpixel(x,y).dword==bcolor.dword)
            {
            putpixel(x,y);
            xx[n]=x;
            yy[n]=y;
            n++; K++; f=j;
            breaK;
            }
        }
    if(i>=8)
        {
        n--;K--;
        if(K<0)
            goto RETU;
        }
    if(n>=maxxyn)  //边界点数量超出范围
        goto RETU;
    if(xx[K]==xx[0]&&yy[K]==yy[0])  //搜索完毕
        breaK;
    }
/*  以下，从所有边界点中确定出交点，并存入交点链表  */
for(i=1;i<n;i++)
    if(yy[i]!=yy[0])
        {
        cK=i;
        breaK;
        }
for(i=n-2;i>0;i--)
    if(yy[i]!=yy[0])
        {
        pK=i;
        breaK;
        }
if((yy[pK]>yy[pK+1]&&yy[cK]>yy[cK-1])||(yy[pK]<yy[pK+1]&&yy[cK]<yy[cK-1]))
    {
    lfill.inspole(xx[pK+1],yy[pK+1]);
```

```
    lfill.inspole(xx[cK-1],yy[cK-1]);
    }
else
    lfill.inspole(xx[cK-1],yy[cK-1]);

for(i=cK+1;i<n;i++)
    {
    if(yy[i]!=yy[cK])
        {
        if((yy[i]>yy[cK]&&yy[cK-1]>yy[cK])||(yy[i]<yy[cK]&&yy[cK-1]<yy[cK]))
            {
            lfill.inspole(xx[cK],yy[cK]);
            lfill.inspole(xx[i-1],yy[i-1]);
            }
        else
            lfill.inspole(xx[cK],yy[cK]);
        cK=i;
        }
    }

lfill.draw();
RETU:
free(xx);free(yy);
}
```

9.5.5 区域填充程序使用示例

示例程序9-2

```
void mian()
{
_640_480_256 A;
A.init();
A.setcolor(7);
A.circlefill(100,100,50);  //画一个实圆
getch();
int *b1[10]={320,0,  480,80,  400,240,  240,240,  160,80};
A.polyfill(5,b1);  //画一个实多边形；
getch();
A.cls0();
int *b2[]={3,  100,0, 150,100, 50,100,  3, 100,20, 130,80, 70,80,  0}
A.polyfill(b2);  //填充两个多边形，得到一个带空洞的图形
getch();
A.cls0();
int i;
/*  以下循环显示10种预定义填充模式的图案  */
for(i=1;i<11;i++)
    {
    A.setfillstyle(i);  //设置填充模式
```

```
    A.bar(i*50,100,i*50+40,200);  //画出并以当前填充模式填充一个矩形
    }
getch();
A.cls0();
unchar m[8]={0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18};
//自定义填充模式的掩码。宽度为两个象素点，间隔6个点的竖线
A.setfillpattern(m);  //设置自定义填充模式
A.bar(0,0,A.WIDE-1,A.HIGH-1);  //以自定义填充模式画满全屏
getch();
A.cls0();
A.poly(5,b1);  //画一个多边形
A.setcolor(4);
A.fillarea(320,100,A.setcolorto(7));
//以任意区域填充方式填充刚画出的多边形，填充颜色为4，填充模式为自定义模式
getch();
A.close();
}
```

9.6　块　操　作

9.6.1　概述

块操作包含两个相互关联的功能，一是，读出并保存一个屏幕矩形块，二是，在原处或在其它地方恢复所读出的矩形块。在程序中，通过调用读扫描线函数来实现块保存功能，通过调用写扫描线函数来实现块恢复功能，因此块操作功能的实现与图形模式无关。

所读出的矩形块需要用一定的存储空间来保存，所需存储空间的大小取决于所读块的大小、分辨率及颜色数据的长度。在那些分辨率较高、颜色数据长度较长的图形模式下，往往需要很大的存储空间来保存所读出的块，如在1024×768×256色模式下，保存1/4的屏幕约需196K的空间，保存1/2的屏幕约需384K，保存全屏则需768K，这时就不能单纯依赖常规内存来保存所读出的块，因此，程序同时实现了用常规内存、扩展内存及硬盘来保存屏幕矩形块的功能，在程序中分别针对这三种存储区域来实现块操作功能，每种存储区域对应于一组块操作函数。

常规内存、扩展内存、硬盘这三种存储区域的容量一个比一个大，但它们的速度也正好一个比一个慢。在应用程序中，通常希望在容量有保证的前提下能获得尽量快的速度，而应用程序的运行环境及它本身的状态往往都是不固定，这就很难预先估计出每种存储区域的可用容量，也就无法事先确定每次的块操作到底应使用那一存储区域，这就要求块操作程序能对各种存储区域的容量进行检测，在应用程序所指定的存储区域容量不够的情况下，能自动转到满足容量要求的其它存储区域。因此程序提供了一组块操作的接口函数，用来统一调用各种存储区域的块操作功能，该组接口函数负责进行容量检测和存储区域的自动转换。

9.6.2　使用常规内存的块操作程序

用常规内存进行块操作时，需由应用程序来申请一块内存，用于保存所读出的数据。所需申请的内存块的大小，由一个专门的函数来获得。

块保存函数仅仅保存矩形块的图象数据，而不保存矩形块的位置及大小，这些数据应由应用程序来保存。在恢复所读出的块时，应用程序必须再次给出矩形块的位置及大小，所给位置可以与保存时的不同，但大小必须相同。使用其它两种存储区域的块操作函数也都是如此。

系统程序9-9 VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：读屏幕矩形块，数据存于常规内存
输入参数： x1＝矩形块左上角的横坐标
           y1＝矩形块左上角的纵坐标
           x2＝矩形块右下角的横坐标
           y2＝矩形块右下角的纵坐标
           buf＝存储所读出数据的内存块的指针
返回值：无
*************************************************************************/
void VGABASE::getimageMEM(int x1,int y1,int x2,int y2,void *buf)
{
char *cp;
int xn,ssize,y;

xn=x2-x1+1;
ssize=scanlinesize(x1,x2);
cp=(char *)buf;
for(y=y1;y<=y2;y++)
    {
    getscanline(x1,y,xn,cp);  //调用读扫描线函数
    cp += ssize;
    }
}

/*************************************************************************
功能：从常规内存恢复屏幕矩形块
输入参数： x1＝目标位置左上角的横坐标
           y1＝目标位置左上角的纵坐标
           xn＝以象素点计的矩形块的宽度
           yn＝以象素点计的矩形块的高度
           buf＝矩形块数据指针
返回值：无
*************************************************************************/
void VGABASE::putimageMEM(int x1,int y1,int xn,int yn,void *buf)
{
int ssize,y,i;
char *cp;

cp=(char *)buf;
ssize=scanlinesize(x1,x1+xn-1);
y=y1;
```

```
for(i=0;i<yn;i++)
    {
    putscanline(x1,y,xn,cp);  //调用写扫描线函数
    y++;
    cp += ssize;
    }
}

/*************************************************************************
功能：取保存矩形块所需缓冲区的大小
输入参数：  x1＝矩形块左上角的横坐标
            y1＝矩形块左上角的纵坐标
            x2＝矩形块右下角的横坐标
            y2＝矩形块右下角的纵坐标
返回值：long＝以字节计的缓冲区的大小
    所需缓冲区的大小与存储区域的类型无关。
*************************************************************************/
long VGABASE::imagesize(int x1,int y1,int x2,int y2)
{
long size;
int K;
if(y1>y2)
    { K=y1; y1=y2; y2=K; }
size=(long)scanlinesize(x1,x2)*((long)y2-(long)y1+1L);
return(size);
}
```

9.6.3　使用扩展内存的块操作程序

由块保存函数来完成扩展内存的分配，应用程序无需进行扩展内存的分配，但在调用块保存函数之前应用程序必须确认有足够的扩展内存可供使用。应用程序必须接收由块保存函数所返回的一个XMS对象的指针，以在块恢复时使用，使用完后应用程序应删除该XMS对象，以释放相应的扩展内存。

系统程序9-10　VGADRAW.CPP

```
-------------------------------------------------------------------------
/*************************************************************************
功能：读屏幕矩形块，数据存于扩展内存
输入参数：  x1＝矩形块左上角的横坐标
            y1＝矩形块左上角的纵坐标
            x2＝矩形块右下角的横坐标
            y2＝矩形块右下角的纵坐标
返回值：XMS*＝对应于一块扩展内存的XMS类对象的指针
*************************************************************************/
XMS *VGABASE::getimageXMS(int x1,int y1,int x2,int y2)
{
int Ksize,ssize,xn,y;
void *buf;
```

```
long offset=0L;
XMS *xms;

xn=x2-x1+1;
Ksize=(int)((imagesize(x1,y1,x2,y2)+1023L)/1024L );
xms=new XMS(Ksize);  //申请扩展内存
ssize=scanlinesize(x1,x2);
buf=malloc(ssize);
for(y=y1;y<=y2;y++)
    {
    getscanline(x1,y,xn,buf);  //首先将一条扫描线读入常规内存
    xms->put((void *)offset,buf,ssize);  //然后从常规内存放入扩展内存
    offset += ssize;
    }
free(buf);
return(xms);
}

/*************************************************************************
功能：从扩展内存恢复屏幕矩形块
输入参数：  x1＝目标位置左上角的横坐标
            y1＝目标位置左上角的纵坐标
            xn＝以象素点计的矩形块的宽度
            yn＝以象素点计的矩形块的高度
            xms＝对应于扩展内存中的一个数据块
返回值：无
*************************************************************************/
void VGABASE::putimageXMS(int x1,int y1,int xn,int yn,XMS *xms)
{
int ssize,y,i;
void *buf;
long offset=0L;

ssize=scanlinesize(x1,x1+xn-1);
buf=malloc(ssize);
y=y1;
for(i=0;i<yn;i++)
    {
    xms->get(buf,(void *)offset,ssize);  //首先将扩展内存的数据读到常规内存
    putscanline(x1,y,xn,buf);  //然后写到屏幕上
    y++;
    offset += ssize;
    }
free(buf);
}
```

9.6.4　使用硬盘的块操作程序

块保存函数在当前盘的当前目录上创建一个文件，将所读出的屏幕数据保存在该文件中，并将指向该文件名字符串的指针返回给应用程序，应用程序必须接收该指针，以在块恢复时使用，使用完后应用程序应删除该文件并释放该指针。程序没有保存文件的完整路径名，因此块保存和块恢复时应保持有相同的当前磁盘路径。

系统程序9-11　VGADRAW.CPP

```
--------------------------------------------------------------------------
/*************************************************************************
功能：读屏幕矩形块，数据存于硬盘文件
输入参数：x1＝矩形块左上角的横坐标
          y1＝矩形块左上角的纵坐标
          x2＝矩形块右下角的横坐标
          y2＝矩形块右下角的纵坐标
返回值：char*＝保存屏幕数据的文件名字符串指针
*************************************************************************/
char *VGABASE::getimageHD(int x1,int y1,int x2,int y2)
{
static int No=0;
char *filename;
FILE *fp;
void *buf;
int ssize,xn,y;

filename=(char *)calloc(13,1);
strcpy(filename,"image_a.swp");
filename[6]+=No;
No++;

xn=x2-x1+1;
ssize=scanlinesize(x1,x2);
buf=malloc(ssize);
fp=fopen(filename,"wb");
for(y=y1;y<=y2;y++)
    {
    getscanline(x1,y,xn,buf);  //首先将扫描线读入常规内存
    fwrite(buf,ssize,1,fp);  //然后写入磁盘文件
    }
free(buf);
fclose(fp);
return(filename);
}

/*************************************************************************
功能：从磁盘文件恢复屏幕矩形块
输入参数：  x1＝目标位置左上角的横坐标
            y1＝目标位置左上角的纵坐标
            xn＝以象素点计的矩形块的宽度
```

```
            yn＝以象素点计的矩形块的高度
            xms＝指向保存屏幕数据的磁盘文件名的指针
返回值：无
*************************************************************************/
void VGABASE::putimageHD(int x1,int y1,int xn,int yn,char *filename)
{
FILE *fp;
void *buf;
int ssize,y,i;

ssize=scanlinesize(x1,x1+xn-1);
buf=malloc(ssize);
fp=fopen(filename,"rb");
y=y1;
for(i=0;i<yn;i++)
    {
    fread(buf,ssize,1,fp);  //首先将文件数据读入常规内存
    putscanline(x1,y,xn,buf);  //然后写到屏幕上
    y++;
    }
fclose(fp);
free(buf);
}
```

9.6.5 块操作的统一接口

这里实现了块操作的一组接口函数，该组接口函数用于进行存储区域的自动转换，并完成对矩形块位置和大小的保存。通过这组接口函数，应用程序能以更为简便的方式进行块操作。

程序中首先定义了一个结构，该结构用于保存所读出的一个屏幕矩形块的如下信息：所使用的存储区域、于每种存储区域相对应的数据指针、矩形块的位置及大小。同时定义了一个枚举(enum)以提供指定存储区域的助记符。

系统程序9-12 VGABASE.H

```
-------------------------------------------------------------------------
enum WHERE {
    inNONE  = 0,  //失败
    inMEM = 1,  //存于常规内存
    inXMS = 2,  //存于扩展内存
    inHD  = 3  //存于硬盘
    };

struct IMAGE {
    int where;  //所使用的存储区域类型
    union IMGhandle {
        void *mem;
        class XMS *xms;
        char *filename;
```

```
        } handle;  //数据块句柄
    int x1,y1,xn,yn;  //矩形块的位置及大小
    ~IMAGE();
    } ;

/*  IMAGE的析构函数，用于释放各种存储区域中的数据块  */
IMAGE::~IMAGE()
{
if(where==inMEM)
    free(handle.mem);
else if(where==inXMS)
    delete handle.xms;
else if(where==inHD)
    {
    remove(handle.filename);
    free(handle.filename);
    }
where=noIN;
}
```

块保存函数返回一个IMAGE结构的指针，应用程序必须接收该指针以在块恢复时使用。程序实现了两个块恢复函数，一个用于将块恢复在原处，一个用于将块恢复在一个新的位置。进行块恢复时，应用程序不再需要给定块的大小，当在原处恢复时也不需要给定块的位置。

系统程序9-12　VGADRAW.CPP

```
/*************************************************************************
功能：读出并保存屏幕矩形块
输入参数：  x1＝矩形块左上角的横坐标
            y1＝矩形块左上角的纵坐标
            x2＝矩形块右下角的横坐标
            y2＝矩形块右下角的纵坐标
            where＝存储位置。该参数可以缺省，缺省时为1
                  1(inMEM)  存于常规内存
                  2(inXMS)  存于扩展内存
                  3(inHD)   存于硬盘
返回值：IMAGE*＝所读出块的有关信息数据的指针
*************************************************************************/
IMAGE *VGABASE::getimage(int x1,int y1,int x2,int y2,int where)
{
int i;
long size;
IMAGE *img;

img=new IMAGE;
if(x1>x2)
    { i=x1; x1=x2; x2=i; }  //使x1<x2
```

```
if(y1>y2)
    { i=y1; y1=y2; y2=i; }  //使y1<y2
img->x1=x1;
img->y1=y1;
img->xn=x2-x1+1;
img->yn=y2-y1+1;
size=imagesize(x1,y1,x2,y2);  //获取所需缓冲区的大小

if(where==inMEM)  //若指定存于常规内存
    {
    long l;
    l=coreleft();  //获得常规内存的可用量
    if(size>=l || size>=0xffffL)  //若不能使用常规内存
        where=inXMS;  //转到使用扩展内存
    else  //可以使用常规内存
        {
        img->handle.mem=malloc((size_t)size);  //申请一块常规内存
        getimageMEM(x1,y1,x2,y2,img->handle.mem);  //调用常规内存块保存函数
        }
    }
if(where==inXMS)  //若指定或转到使用扩展内存
    {
    if(XMS::OK==0)  //若扩展内存不存在
        where=inHD;  //转到使用硬盘
    else  //若扩展内存存在
        {
        long l=0;
        l=(long)XMS::largestblocK()-1L;  //取最大的可用扩展内存块的大小
        l *= 1024L;
        if( size>=l )  //若扩展内存空间不够
            where=inHD;  //转到使用硬盘
        }
    if( where==inXMS )  //若仍然使用扩展内存
        img->handle.xms=getimageXMS(x1,y1,x2,y2);  //调用扩展内存块保存函数
    }
if(where==inHD)  //若指定或转到使用硬盘
    img->handle.filename=getimageHD(x1,y1,x2,y2);  //调用硬盘块保存函数
img->where=where;
return(img);
}

/*************************************************************************
功能：在一个新的位置恢复屏幕矩形块
输入参数：  x1＝目标位置左上角的横坐标
            y1＝目标位置左上角的纵坐标
            img＝所要恢复数据的信息结构指针
返回值：无
*************************************************************************/
```

```
void VGABASE::putimage(int x1,int y1,IMAGE *img)
{
if( img->where==inMEM )
    putimageMEM(x1,y1,img->xn,img->yn,img->handle.mem);
else if( img->where==inXMS )
    putimageXMS(x1,y1,img->xn,img->yn,img->handle.xms);
else if( img->where==inHD )
    putimageHD(x1,y1,img->xn,img->yn,img->handle.filename);
}

/*************************************************************************
功能：在原处恢复屏幕矩形块
输入参数：img＝所要恢复数据的信息结构指针
返回值：无
*************************************************************************/
void VGABASE::putimage(IMAGE *img)
{
if( img->where==inMEM )
    putimageMEM(img->x1,img->y1,img->xn,img->yn,img->handle.mem);
else if( img->where==inXMS )
    putimageXMS(img->x1,img->y1,img->xn,img->yn,img->handle.xms);
else if( img->where==inHD )
    putimageHD(img->x1,img->y1,img->xn,img->yn,img->handle.filename);
}
```

9.6.6　块操作使用示例

示例程序9-3

```
void main()
{
_1024_768_256 A;
A.init();
XMS::init();  //将XMS管理程序初始化
A.setcolor(1);
A.cls();
void *buf;
buf=malloc(imagesize(0,0,20,20));
A.getimageMEM(0,0,20,20,buf);  //使用常规内存的块保存
XMS *xms;
xms=A.getimageXMS(30,30,300,300);  //使用扩展内存的块保存
char *filename;
filename=A.getimageHD(310,310,760,760);  //使用硬盘的块保存
getch();
A.cls0();
A.putimageMEM(0,0,21,21,buf);  //使用常规内存的块恢复
A.putimageMEM(40,0,21,21,buf);  //恢复到一个新的位置
free(buf);  //释放常规内存块
A.putimageXMS(30,30,271,271,xms);  //使用扩展内存的块恢复
```

```
delete xms;  //释放扩展内存块
A.putimageHD(310,310,451,451,filename);  //使用硬盘的块恢复
remove(filename);  //删除保存屏幕数据的磁盘文件
free(filename);  //释放文件名字符串
getch();
IMAGE *img;
img=A.getimage(0,0,400,400,inXMS);  //用接口函数进行块保存，指定存于扩展内存
A.cls0();
A.putimage(300,300,img);  //用接口函数进行块恢复，恢复到一个新的位置
delete img;  //删除，以释放所使用的存储空间
getch();
A.close();
    }
```

第 10 章　VGA图形显示小结

10.1　图形显示程序使用方式

从第3章至第9章，本书实现了一套完整的VGA图形显示程序，本章介绍如何在应用程序中使用这套程序。对于一般的结构化程序设计语言来说，使用一组基础程序的基本方式就是调用，即调用一个函数或调用一个子程序，而对于C++来说，则还存在着另外一种使用方式——继承。这里只介绍如何在应用程序中调用这套VGA图形显示程序。继承是一种更为灵活也更为复杂的程序使用方式，在下一节及以后的各章中将针对一些具体问题介绍如何通过继承来使用这套VGA基础图形显示程序。

10.1.1　基本使用方式

在应用程序中使用这套图形显示程序的基本步骤如下：

①根据所要使用的图形模式选定一个图形模式类，定义该类的一个对象，记为A；

②调用A.init()，这时屏幕显示模式被设置为所选定的图形模式；

③若在程序中要使用块操作功能，调用XMS::init()，初始化XMS管理程序；

④通过对象A调用各个图形显示函数，进行应用程序的图形显示；

⑤在程序退出前调用A.close()，以关闭图形模式，返回标准的字符显示模式。

下面给出示例程序10-1，该程序在屏幕左上角显示一个实矩形，然后通过块操作将该实矩形拷贝到屏幕右下角，所使用的图形模式为1024×768×256色。

示例程序10-1

```
--------------------------------------------------------------------------
void main()
{
_1024_768_256 A;
A.init();
XMS::init();
A.setcolor(1);
A.bar(0,0,200,200);
IMAGE *img;
img=A.getimage(0,0,200,200,inXMS);
A.putimage(A.WIDE-201,A.HIGH-201,img);
delete img;
getch();
A.close();
}
--------------------------------------------------------------------------
```

10.1.2　交替使用多种图形模式

在某些情况下，一个应用程序中可能需要交替使用多种图形模式，本书所实现的这套图形显示程序能够支持这种使用方式。下面给出的示例程序10-2交替使用了三种图形模式，可在每种图形模式下画一个实矩形。

示例程序10-2

```
--------------------------------------------------------------------------
void main()
{
_640_480_16 A;
_1024_768_256 B;
_320_200_16M C;
A.init();  //设置为640×480×16色模式
A.bar(100,100,200,200,A.setcolorto(1));
getch();
B.init();  //切换到1024×768×256色模式
B.bar(100,100,200,200,A.setcolorto(1));
getch();
C.init();  //切换到320×200×16M色模式
C.bar(100,100,200,200,C.setcolorto(0,0,127));
getch();
C.close();  //关闭图形显示，返回到标准字符显示模式
}
--------------------------------------------------------------------------
```

10.1.3 图形模式的动态设置

某些应用程序可能需要在运行时刻根据用户的选择或运行环境来确定其工作的图形模式，这时可通过对图形模式类对象的动态定义来实现图形模式的动态设置，下面给出示例程序10-3，可从4种图形模式中动态地选择一种图形模式。

示例程序10-3

```
--------------------------------------------------------------------------
void main()
{
VGABASE *A;
int mode;  //存放所要采用的图形模式的编号
mode=getmode();
//这是由应用程序定义的一个函数，其根据用户选择或运行环境确定出所要采用的图形模式
/*  以下条件语句实现图形模式的动态选择  */
if(mode==1)
    A=new _640_480_16;
else if(mode==2)
    A=new _1024_768_16;
else if(mode==3)
    A=new _640_480_256;
else if(mode==4)
    A=new _1024_768_256;
A->init();
appfun();  //完成应用程序图形功能的函数
A->close();
delete A;  //对象是动态建立的，用完后需要删除
}
--------------------------------------------------------------------------
```

10.2　扩展到新的图形模式

在本书的1.3.5节中，将VGA的图形显示模式分为普通模式和高级模式两类，目前,那些高级模式都需要高性能的显示卡和显示器来支持，编写这些模式下的程序需要有相应的设备来支持模式参数的测试和程序的调试，因此本书没有实现对这些高级模式的支持，而只提供了对18种普通图形模式的支持，但很容易将本书的程序扩展到对那些高级模式提供支持，下面介绍如何进行这种扩展。

10.2.1　参数检测

要在程序中引入一种新的图形模式，首先要解决的一个问题就是要获得这种图形模式的有关参数，所需获得的参数包括：模式号、每条扫描线所占字节数、所占用的显示存储器页数、屏幕水平象素数、屏幕垂直象素数。如果所引入的模式属于一种新的色彩模式，则还需搞清该色彩模式下的显示存储器结构，这包括：是否采用了位面技术或有几个位面、每个象素点所占字节数或位数、象素数据的格式。对那些较新的图形模式来说，要想直接从有关资料中获取这些参数和情况往往是比较困难的，本书附带的软盘提供了一个实用程序TVESA.EXE，该程序能够检测出为当前显示卡所支持的每种显示模式的上述有关参数，下面介绍如何通过该程序来获得这些参数。

TVESA.EXE程序首先通过调用VESA BIOS功能0获得当前显示卡所支持的每一种显示模式的模式号，然后调用VESA BIOS功能1，获得每一模式号所对应显示模式的有关信息，对所获得的信息进行一定的处理后输出到屏幕上。下面说明其所输出的有关部分信息。

Mode No. : VESA模式号
Graphics mode : 水平象素数×垂直象素数×颜色数
Bytes of scanline : 每条扫描线在每个位面所占字节数
Memory Size : 所占用的显示存储器容量
Number of Planes : 位面数
Bits of pixel : 每个象素所占位数

将Memory Size除以64K再除以位面数即可得到所占用的显示存储器页数，其它参数则可直接使用。

下面列出由该程序所检测到的两种图形模式的结果。

```
--------------------------------------------------------------------------------
    Mode No. : 104H
    Graphics mode  1024 * 768 * 16
    Bytes of scanline : 128 Bytes
    Memory Size : 393216 Bytes      384K
    Number of Planes : 4
    Bits of pixel : 4
--------------------------------------------------------------------------------
    Mode No. : 112H
    Graphics mode : 640 * 480 * 16777216
    Bytes of scanline : 2048 Bytes
```

```
    Memory Size : 983040 Bytes      960K
    Number of Planes : 1
    Bits of pixel : 24
------------------------------------------------------------------------------
```

该程序不能给出象素数据的格式，只在直接色彩模式下需要确定象素数据的格式，这只能通过试验得到，在已经获得其它参数的情况下，这项试验工作是比较简单的，通常的做法是，首先设想一种数据格式，按此编写一个写点程序，然后根据所得到的屏幕象素点的颜色与预计颜色之间的差别对程序进行修改，经过几次修改之后一般就能得到正确的象素数据格式。

10.2.2 支持256色、真彩色、高彩色的新模式

当所要支持的新图形模式属于某种已有的色彩模式时，程序的扩展工作就非常简单，这时只需从相应的色彩模式类中派生出一个新的图形模式类，然后根据所获得的有关参数定义出该类的构造函数，即可完整实现对这种新图形模式的支持。

这里针对一种具体的新图形模式1280×1024×256色模式，来讨论如何将程序扩充到对这种新的图形模式提供支持。该模式属于256色模式，因此从VGA256类中派生出一个新类_1280_1024_256，然后定义该类的构造函数，并在其中进行参数设置。已获得该模式下的有关参数如下：VESA模式号为107H，每条扫描线所占字节数为1280，占用的显示存储器页数为20，屏幕水平象素数为1280，屏幕垂直象素数为1204。下面示例程序10-4完整实现了对这种新图形模式的支持。

示例程序10-4

```
------------------------------------------------------------------------------
class _1280_1024_256 : public VGA256
    {
public:
    _1280_1024_256();
    };

_1280_1024_256::_1280_1024_256()
{
SCANLENG=1280;
WIDE=1280;
HIGH=1024;
PAGEN=20;
VESAmodeNo=0x0107;
}
------------------------------------------------------------------------------
```

只需在程序中加入上述10余行代码，即可获得这种图形模式下的所有功能，包括图形操作功能、绘图功能以及本书后面所述的字符显示功能、鼠标操作功能等，程序这种良好的可扩充性主要得益于C++。

上面给出的这种程序扩展方式对256色、真彩色、高彩色的新图形模式是完全有效的，但不能完全适用于16色的新图形模式。

10.2.3　支持16色的新模式

已实现的16色模式下的图形操作函数是按照行外分页的情况编写的，而新的16色图形模式极有可能出现行内分页的情况，这时就不能按上面的方法简单的进行程序扩充，而必须重写有关的图形操作函数，所需重写的函数包括画扫描线、读扫描线、写扫描线3个函数，在行外分页和行内分页的情况下点操作函数及清屏函数是没有差别的，因此它们不需要重写。

当要支持一种具有行内分页情况的新16色模式时，可按两种方式重写扫描线操作函数，一是改写源程序，即直接重写VGA16类中的3个函数VGA16::scanline()、VGA16::getscanline()及VGA16::putscanline()。二是从VGA16类中派生一个新类，在这个新类中重写上述3个函数，这时就不需要修改已有的源程序。下面按照第二种方式，针对一个具体的新图形模式1280×1024×16色模式，介绍程序扩充的方法。

首先从VGA16类派生一个新类VGA16A，在VGA16A中重新定义各扫描线操作函数，然后从VGA16A派生一个图形模式类_1280_1024_16，并定义该类的构造函数。下面给出具体的示例程序10-5。

示例程序10-5

```
--------------------------------------------------------------------------
class VGA16A : public VGA16
    {
public:
    void scanline(int x1,int x2,int y);
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);
    void scanline(int x1,int x2,int y,COLOR color)
        { CUR_COLOR=color; scanline(x1,x2,y); }
    };

class _1280_1024_16 : public VGA16A
    {
public:
    _1280_1024_16();
    };

void VGA16A::scanline(int x1,int x2,int y)
{
……  //按行内分页的情况实现画扫描线功能
}

void VGA16A::getscanline(int x1,int y,int n,void *buf)
{
……  //按行内分页的情况实现读扫描线功能
}

void VGA16A::scanline(int x1,int y,int n,void *buf)
{
……  //按行内分页的情况实现写扫描线功能
}
```

```
_1280_1024_16::_1280_1024_16()
{
SCANLENG=160;  //每条扫描线在每个位面中所占字节数
WIDE=1280;  //屏幕水平象素数
HIGH=1024;  //屏幕垂直象素数
PAGEN=3;  //所占显示存储器页数
VESAmodeNo=0x0106;  //VESA模式号
}
```

10.2.4 支持新的色彩模式

如果所要支持的新图形模式不属于已有的4种色彩模式，如早期的4色、2色模式或以后可能出现的32位色模式，这时就需首先从VGABASE类中派生出一个新的色彩模式类，在该类中定义这种新色彩模式下的所有图形操作函数，然后再从该类派生出一个图形模式类，并定义其参数设置函数。下面针对一种假设的320×200×32位色模式介绍如何实现对一种新的色彩模式的支持。

首先从VGABASE类派生出一个32位(4G)色模式类VGA4G，在该类中定义32位色模式下的所有图形操作函数，然后从VGA4G派生出一个图形模式类_320_200_4G，并定义该类的构造函数。下面给出具体的示例程序10-6。

示例程序10-6

```
class VGA4G : public VGABASE
    {
public:
    void setcolor(unchar r,unchar g,unchar b);
    COLOR setcolorto(unchar r,unchar g,unchar b);
    void putpixel(int x,int y);
    union COLOR getpixel(int x,int y);
    void cls(void);
    int scanlinesize(int x1,int x2);
    void scanline(int x1,int x2,int y);
    void getscanline(int x1,int y,int n,void *buf);
    void putscanline(int x1,int y,int n,void *buf);

    void putpixel(int x,int y,COLOR color)
        { CUR_COLOR=color; putpixel(x,y); }
    void scanline(int x1,int x2,int y,COLOR color)
        { CUR_COLOR=color; scanline(x1,x2,y); }
    void cls(COLOR color)
        { CUR_COLOR=color; cls(); }
    };

class _320_200_4G : public VGA4G
    {
public:
```

```
    _320_200_4G();
    };

void VGA4G::setcolor(unchar r,unchar g,unchar b)
{
……
}

COLOR VGA4G::setcolorto(unchar r,unchar g,unchar b)
{
……
}

void VGA4G::putpixel(int x,int y)
{
……
}

COLOR VGA4G::getpixel(int x,int y)
{
……
}

void VGA4G::cls(void)
{
……
}

int VGA4G::scanlinesize(int x1,int x2)
{
……
}

void VGA4G::scanline(int x1,int x2,int y)
{
……
}

void VGA4G::getscanline(int x1,int y,int n,void *buf)
{
……
}

void VGA4G::putscanline(int x1,int y,int n,void *buf)
{
……
}
```

```
_320_200_4G::_320_200_4G()
{
SCANLENG=1280;
WIDE=320;
HIGH=200;
PAGEN=4;
VESAmodeNo=0x????;
}
```

10.3 显示速度测试

这里介绍本书所实现的VGA基础图形显示程序在各种显示模式下的速度如何进行测试，可以采用相同的方式对Broland C++的图形函数进行测试。

10.3.1 测试对象、环境及项目

测试对象为图形显示程序所支持的18种图形模式中的14种，15位色模式和16位色模式下的显示速度是完全一样的，因此去掉了4种15位色模式，在640×480×16色模式下还对Borland C++的图形函数进行了测试。

分别在两种硬件环境下进行测试，第一种环境的配置为：486DX2/66CPU，8K外部Cache，VESA局部总线，Trident 9400 32位VESA局部总线显示卡，1M显示存储器，其能支持所要测试的所有显示模式。第二种环境的配置为：386DX/40CPU，64KCache，ISA总线，TVGA9000显示卡，512K显示存储器，其只能支持3种16色模式及前4种256色模式。按通常的主机测速软件，第一种环境的主机速度为第二种的4倍。

所测试的速度有3个：写单个象素点的速度，称单点速度，单位为每秒所写千个象素点数；写满整个屏幕的速度，称整屏速度，单位为每秒所写满的屏幕数；有效数据传输速度，称数据传输率，单位为每秒所传输的K字节数。分别用写点函数和画扫描线函数来测试这3个速度，共给出6组测试结果。

10.3.2 测试方式及程序

所采用的测试方式为：用写点函数或画扫描线函数写满整个屏幕若干次，由主机系统所维持的每秒18.2次的计数测出所用时间，根据写满屏的次数及所用时间算出整屏速度，然后根据当前图形模式下屏幕象素点数算出单点速度，再根据每个象素点的数据长度算出数据传输率。具体的测试程序如应用程序10-1所示。

应用程序10-1 VGASPEED.CPP

```
#include <graphics.h>
#include <stdio.h>
#include <bios.h>

#include "\vga\vga.h"

#define ND 10  //写点操作写满屏幕的次数
```

```
#define NL 100  //画扫描线操作画满屏幕的次数

class Speed {
    long start_dida,end_dida,dida_num;  //开始时间、结束时间、时程
    double dot_s,scr_s,dat_s;  //单点速度、整屏速度、数据传输率
    int wide,high,dot_bit; //水平象素数、垂直象素数、每个象素数据的位数
public:
    Speed(int *p);  //构造函数，确定有关参数
    void start();  //记录开始时间
    void end();  //记录结束时间
    void calu(int n);  //计算速度
    void out();  //输出
    } ;

Speed::Speed(int *p)
{
wide=p[0]; high=p[1]; dot_bit=p[2];
start_dida=end_dida=_dida_num=0;
}

/*  记录开始的时间  */
void Speed::start()
{
start_dida=biostime(0,start_dida);
}

/*  记录结束的时间  */
void Speed::end()
{
end_dida=biostime(0,end_dida);
dida_num=end_dida-start_dida;
}

/*  速度计算  */
void Speed::calu(int scr_n)
{
double dida=18.2;
double d1,d2,d3;
d1=(double)wide*(double)high;
d2=dida_num; d2 /= dida;
d1 /= d2;
d1 *= (double)scr_n;
dot_s=d1/1000.0;  //单点速度
scr_s=(double)scr_n/d2;  //整屏速度
d3=(double)dot_bit/8.0;
dat_s=(d1/1024.0)*d3;  //数据传输率
}
```

```
void Speed::out()
{
printf("%4d*%3d*%2d %5.0f %5.2f %5.0f   ",wide,high,dot_bit,dot_s,scr_s,dat_s);
}

/*  用写点操作写满屏幕ND次  */
void run1(Speed *sw)
{
int i,j,K;
int wide=vga->WIDE;
int high=vga->HIGH;
sw->start();
for(i=0;i<ND;i++)
    for(j=0;j<high;j++)
        for(K=0;K<wide;K++)
            vga->putpixel(K,j);
sw->end();
}

/*  用画扫描线操作画满屏幕NL次  */
void run2(Speed *sw)
{
int i,j,K;
int wide=vga->WIDE-1;
int high=vga->HIGH-1;
sw->start();
for(i=0;i<NL;i++)
    for(j=0;j<=high;j++)
        vga->scanline(0,wide,j);
sw->end();
}

#define N 15

void main()
{
int p[N][3]={{640,480,4},{800,600,4},{1024,768,4},
            {320,200,8},{640,400,8},{640,480,8},{800,600,8},{1024,768,8},
            {320,200,16},{512,480,16},{640,480,16},{800,600,16},
            {320,200,24},{640,480,24},
            {640,480,4}};
Speed sp1[N]={p[0],p[1],p[2],p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],
            p[11],p[12],p[13],p[14]};
Speed sp2[N]={p[0],p[1],p[2],p[3],p[4],p[5],p[6],p[7],p[8],p[9],p[10],
            p[11],p[12],p[13],p[14]};

int i,j,K;
for(i=0;i<N-1;i++)
```

```
    {
    switch(i) {
        case 0: vga=new _640_480_16; breaK;
        case 1: vga=new _800_600_16; breaK;
        case 2: vga=new _1024_768_16; breaK;
        case 3: vga=new _320_200_256; breaK;
        case 4: vga=new _640_400_256; breaK;
        case 5: vga=new _640_480_256; breaK;
        case 6: vga=new _800_600_256; breaK;
        case 7: vga=new _1024_768_256; breaK;
        case 8: vga=new _320_200_64K; breaK;
        case 9: vga=new _512_480_64K; breaK;
        case 10: vga=new _640_480_64K; breaK;
        case 11: vga=new _800_600_64K; breaK;
        case 12: vga=new _320_200_16M; breaK;
        case 13: vga=new _640_480_16M; breaK;
        }
    if(vga->init()==0)
        continue;
    if(i<=7) vga->setcolor(7);
    else vga->setcolor(127,127,127);
    run1(sp1+i);  //测试写点速度
    sp1[i].calu(ND);
    if(i<=7) vga->setcolor(1);
    else vga->setcolor(63,63,191);
    run2(sp2+i);  //测试画扫描线速度
    sp2[i].calu(NL);
    vga->close();
    delete vga;
    }

int drv=VGA,mode=VGAHI;
registerbgidriver(EGAVGA_driver);
initgraph(&drv,&mode,"");
setcolor(1);
/*  以下6行，测试Borland的写点速度  */
sp1[N-1].start();
for(i=0;i<ND;i++)
    for(j=0;j<480;j++)
        for(K=0;K<640;K++)
            putpixel(K,j,1);
sp1[N-1].end();
sp1[N-1].calu(ND);
/*  以下6行，测试Broland的画扫描线速度  */
sp2[N-1].start();
for(i=0;i<NL;i++)
    bar(0,0,639,479);
sp2[N-1].end();
```

```
sp2[N-1].calu(NL);
restorecrtmode();
/*  以下输出结果  */
for(i=0;i<N;i++)
    {
    sp1[i].out(); sp2[i].out();
    printf("\n");
    }
}
```

10.3.3 测试结果及分析

表10-1给出了速度测试的结果。

表10-1 显示速度测试结果

主机	图形模式	写 点			画扫描线		
		单点速度/千点/s	整屏速度/屏/s	数据传输率/B/s	单点速度/千点/s	整屏速度/屏/s	数据传输率/KB/s
486	640*480*16 Borland C++	95	0.31	46	24894	80.9	12133
	640*480*16	151	0.49	74	21926	71.4	10706
	800*600*16	153	0.32	75	28181	58.7	13760
	1024*768*16	155	0.20	76	28626	36.4	13978
	320*200*256	485	7.58	474	4236	66.2	4136
	640*400*256	475	1.86	464	5449	21.3	5322
	640*480*256	474	1.54	463	5455	17.8	5327
	800*600*256	475	0.99	464	5460	11.4	5332
	1024*768*256	472	0.6	461	5341	6.8	5216
	320*200*64K	466	7.28	910	5295	82.7	10341
	512*480*64K	461	1.88	901	5357	21.8	10462
	640*480*64K	462	1.5	902	4799	15.6	9373
	800*600*64K	462	0.96	903	3142	6.6	6138
	320*200*16M	388	6.07	1138	1820	28.4	5332
	640*480*16M	386	1.26	1130	1076	3.5	3153

续表

主机	图形模式	写　点			画扫描线		
		单点速度/千点/s	整屏速度/屏/s	数据传输率/B/s	单点速度/千点/s	整屏速度/屏/s	数据传输率/KB/s
386	640*480*16 Borland C++	31	0.1	15	6656	21.7	3250
	640*480*16	62	0.21	30	4992	16.3	2438
	800*600*16	62	0.13	30	5050	10.5	2466
	1024*768*16	62	0.08	30	5301	6.7	2588
	320*200*256	149	2.33	146	1099	17.2	1073
	640*400*256	151	0.59	148	1095	4.3	1069
	640*480*256	151	0.49	148	1094	3.6	1068
	800*600*256	153	0.32	149	1390	2.9	1357

以下对测试结果作一些分析。

1．单点速度与整屏速度

单点速度用于对各种图形模式下的显示速度作绝对比较，整屏速度则用于作相对比较。当需要根据速度来选择软件的图形模式时，整屏速度具有更大的参考价值。单点速度主要与色彩模式相关，同一种色彩模式中的各种图形模式都具有基本相同或相近的单点速度。整屏速度同时与色彩模式及分辨率相关，在同一种色彩模式下，整屏速度基本与分辨率成反比。数据传输率与单点速度成正比，在每种色彩模式下这两者之间都具有一个固定的比例系数。各种色彩模式下的数据传输率存在着很大的差别，因此不能根据各种色彩模式下的象素数据长度来估计各种色彩模式间的速度差异。

2．写点与画扫描线

画扫描线的速度要大大高于写点的速度，在大部分图形模式下这两者的速度约相差10倍，在24位色模式下的差别较小，在16色模式下的差别则达到了100倍，在各种色彩模式下，读、写扫描线的速度与画扫描线的速度都基本相近，因此只要有可能就应尽量采用扫描线操作来完成图形显示。扫描线操作的单点速度与扫描线的长度有关，扫描线越长其单点速度就越快，当扫描线的长度为1个象素时，其单点速度就肯定要低于写点操作，两者之间的速度相交点大约为3个象素。

3．Broland C++图形函数的速度

由测试结果可以看到，Broland C++的写点函数比本书的写点函数要慢将近一半，其速度较慢的原因可能出于两个方面：一是可能采用了写方式0而不是写方式2来实现写点操作；二是可能采用了更为保险的寄存器操作策略。这两个方面的原因都会导致速度的明显降低。Broland的画扫描线函数比本书的要快，但差别不是太大，约为15%。

4．16色模式下的写点速度

16色模式下象素点的数据长度是最短的，但其写点速度却最慢，而且慢的幅度很大，其

单点速度约比其它色彩模式慢3倍，数据传输率则慢6～15倍，如果了解了16色模式下的写点原理，就不难理解其中的原因，在16色模式下写一个象素点需要进行一系列的寄存器操作和位屏蔽操作，在这两方面所花费的时间要远远大于最终将象素数据写入显示存储器的时间，而其它模式下都不需要进行寄存器操作和位屏蔽操作。

5．16色模式下的画扫描线速度

16色模式下的写点速度是最慢的，但其画扫描线的速度却出奇地快，16色模式下一条扫描线的数据量是256色模式的一半，但其画扫描线的速度却是256色的4倍，数据传输率是256色的2倍。这种极快的速度主要得益于位面技术，在16色模式下画扫描线时，CPU进行一次8位的写操作就能将预先放在设置重置寄存器中的颜色值写入8个象素点的4个位面中，这就将实际的数据操作量减到了正常数据量的1/4，因而16色模式下画一条扫描线的实际数据操作量只有256色的1/8，但由于需要进行寄存器操作及位屏蔽操作，且写入的数据需经过图形控制器的处理，所以速度没有达到256色的8倍，而只是4倍。

6．高彩色模式下的速度

高彩色的象素数据量是256色的2倍，但其写点速度却与256色基本相同，数据传输率则是256色的2倍，这主要是因为在高彩色模式下都采用了字操作指令来写存储器，而256色模式下采用的是字节操作指令，这两种指令完成一次操作的速度是基本相同的，但所传输的数据量却相差一倍，这就使得高彩色模式具有与256色模式基本相同的显示速度。

7．显示卡对显示存储器的占用

扫描线越长，画扫描线的单点速度就越快，在速度测试中，每次所画扫描线的长度都为当前模式下的屏幕水平象素数，因此在相同的色彩模式下，较高的分辨率应具有较高的画扫描线的单点速度，但在某些色彩模式下，分辨率高到一定程度后画扫描线的单点速度反而会有较大的降低，这主要是因为更高的分辨率会带来更大的显示刷新数据量，也就会增加显示卡对显示存储器的占用时间，从而降低CPU写显示存储器的速度。由于写点操作的时间主要花在地址计算或寄存器操作上，而扫描线操作的时间则主要花在访问显示存储器上，因此显示卡对显示存储器的占用主要对扫描线操作的速度产生影响，而对写点的速度没有影响。在一定的显示卡上，只有显示刷新的数据量大到一定程度后，这种影响才会产生较明显的效果。显示刷新的数据量不仅取决于分辨率及颜色数据的长度，而且还决定于当前显示模式下的帧频及是否采用了隔行扫描。

10.3.4 速度的提高

本书的主要目的是介绍VGA的图形显示原理及技术，而保持程序结构的清晰和简洁将更有益于对原理和技术的介绍，因此在本书的编程中更加注重的是程序的结构，而不是速度，所以本书所实现的这套程序在速度上存在着较大的改进余地，下面介绍可以从哪些方面来提高程序的速度。

1．移植到C语言

C++的采用是影响程序速度的一个主要因素。程序中那些直接决定显示速度的图形操作函数都被说明为虚函数，对虚函数的调用只能在程序运行时动态确定，而不能在编译时预先确定，因此与通常的函数调用相比，对虚函数的调用就需要额外花费一段时间，这段额外花费的时间对程序速度的实际影响取决于其占函数本身运行时间的比例及函数被循环调用的程

度或可能性。对于256色、高彩色、真彩色模式下的点操作函数，动态调用额外所花费的时间约占函数正常运行时间的15%，对16色模式下的点操作函数约为5%，对扫描线操作函数则与扫描线的长度有关，通常为1%左右。可见，虚函数的动态调用主要对点操作的速度产生影响，而本书的大部分绘图函数都是通过循环调用写点函数来实现屏幕显示的，这些绘图函数的速度都要受到与写点函数同等程度的影响。

将程序全部移植到C语言，就可使那些与点操作有关的很大一部分函数的速度提高15%左右，但这将完全改变程序的结构，需要对整套程序重新进行组织，而且所得到的程序的可维护性及可扩充性将明显不如原来的C++程序。当然也可以保留C++而不使用虚函数，但对这套程序来说，这种做法对程序结构的影响与移植到C语言是基本相同的。

2．直接实现绘图函数

程序中的绘图函数都是通过调用有关的图形操作函数来完成图形显示，如果直接在这些函数中实现图形显示将能提高它们的速度。在16色模式之外的色彩模式下，这种做法主要能免去函数调用的时间并能节省地址计算的时间，这不会使速度有太大的提高，但在16色模式下，这种做法能免去大量重复的寄存器操作，而寄存器操作是影响16色模式下显示速度的最主要因素，因而能使速度有成倍的提高。如果只按这种方式实现16色模式下的绘图函数，具体做法是：在16色模式类VGA16中重新说明和定义所要修改的绘图函数。如果要按这种方式实现所有模式下的绘图函数，具体做法是：在VGABASE类中将有关的绘图函数说明为纯虚函数，然后再在各个色彩模式类中分别定义这些函数。按这种方式实现绘图函数将使程序的规模和复杂性有很大的提高，而且会较大地损害程序的可维护性。

3．对不同的分页情况分别编程

本书在每种色彩模式下都只针对最复杂的分页情况进行编程，如果在每种色彩模式下都分别针对每种分页情况进行编程，将能提高那些具有较简单分页情况的图形模式的显示速度，但提高的幅度是非常有限的，在较简单的分页模式下运行较复杂的分页程序，只会多执行1个或2个转跳指令，对扫描线操作而言，2个转跳指令对速度的影响是完全可以忽略的，对于点操作而言其影响也是非常小的。进行这种修改会增加程序量，但对程序的结构没有太大影响。具体做法为：在色彩模式类与图形模式类之间增加一个类层次，在这个层次的多个类中分别针对不同的分页情况实现有关的图形操作函数，最终的图形模式类从这些新增的类中派生。

4．按字进行扫描线操作

在16色、256色及24位色模式下，程序都是按字节进行扫描线操作，如果按字进行操作则能较明显地提高速度，这只是对函数内部的修改，因此不会影响程序的结构，但会影响程序的清晰性。

5．用16位的I/O指令进行寄存器操作

程序都是用 8 位的 I/O 指令进行寄存器操作，用 16 位的 I/O 指令能加快速度，并能简化程序，但某些 VGA 可能不支持这种方式。这只对 16 色模式下的图形操作速度产生影响。其具体实现方式见 2.2 节。

第 11 章　字 符 显 示

字符显示是每个应用程序都需使用的一项最基本功能，在字符模式下，应用程序能直接从硬件或环境软件那里获得比较完善的字符显示功能，而在图形模式下，则很难单纯依赖硬件或环境软件来完成有效的字符显示，这就需要专门编写程序来提供字符显示功能。本章将介绍有关的字符显示技术，并实现一组字符显示的基础支持程序。

11.1　字 库 类 型

在图形系统中，每个字符实际上就是一个具有一定形状的图象，每个字符图象都需要用一定的数据来描述，若干个字符的图象数据的有序集合即构成一个字库。每个字库都对应有一定的字符集、一定的字体和一定的字符大小，并具有一定的字符描述方式和一定的字库结构，下面对这些因素分别进行介绍。

1. 字符集及机内码

一个字库中所包含的字符并不是随意确定的，而是要对应于一定的字符集，我国在计算机上所用到的字符集主要有两个，一是ASCII字符集，包含了所有的英文字母、所有的数字及一些常用的图形符号，该字符集共有256个字符，其中的前128个字符为ASCII基本字符集，后128个为ASCII扩展字符集。二是由我国1981年所公布的国家标准GB2312-80所规定的汉字字符集(**汉字国标字符集**)，其中包含了6763个汉字(**汉字基本字符集**)，700余个西文字母、数字及图形符号(**汉字图形字符集**)。一个汉字库可能只包含了汉字基本字符集(称**基本汉字库**)，也可能只包含了汉字图形字符集(称**图形汉字库**)，也可能两者都包含(称**国标汉字库**)。

一个字符集中的每一个字符在计算机内部都用一定的代码来表示，称为字符的机内码。ASCII字符的机内码通常就称为ASCII码，其长度为1个字节，正好可以表示256个字符。目前所采用的汉字机内码，其长度为2个字节，这两个字节取值范围都为A1h～FEh，可表示8836个字符，这超过了汉字国标字符集所包含的字符数，因此某些机内码在汉字国标字符集中没有定义相对应的字符，它们可留作扩充使用。对汉字图形字符集，两字节机内码的高字节的取值范围为A0h～AFh，其中AAh～AFh在国标字符集中没有定义；对汉字基本字符集，机内码高字节的取值范围为B0h～FEh，其中F8h～FEh在国标字符集中没有定义。

2. 字体

字体指字符显示的风格，英文字符和汉字字符都有着非常丰富的字体，常见的英文字体有：Courier、Roman、Sans Serif、Script、Triplex、Gothic等，最常用的汉字字体有4种：宋体、黑体、楷体、仿宋，其它的汉字字体还有：行楷、隶书、标宋、综艺等，英文图形字符和汉字图形字符则没有字体变化。一个字符的每种字体都具有不同形状，因而需要用不同的数据来描述，一个字库一般只提供一种字体，一个字符集的多种字体需要由多个字库来提供。

3. 字符大小

在字库中一般都是基于象素点或者说基于整数来描述字符的形状，这时所给出的字符

图象就必然具有一定的大小，字符的大小通常用以象素点计的字符的宽度和高度来表示。在汉字字库中，每个字符都具有相同的大小，且字符图象一般都是正方形即宽度等于高度。在英文字库中，各个字符的相对大小则与字体有关，在某些字体中，一个字库中的各个字符的宽度是有差别的，这类字体称为比例字体，而在另一些字体中，各个字符的大小都是相同的，这类字体称为等宽字体。大部分英文字体都属于比例字体，在字符模式下所显示出的字体以及英文打字机上的字体则属于等宽字体。在比例体字库中通常用字符的高度来标识字符的大小。

字库中的字符都有一定的大小，软件可以对字库中的字符进行放大或缩小，对字符的放大或缩小往往会对输出效果产生一定损害，如何保证字符缩放的质量，是目前字库技术所要解决的最主要问题。如果缩放后仍能得到高质量的输出效果，那就只需对一定字符集的某种字体提供一个字库，否则，就需要按不同的大小提供多个字库。

4. 字符描述方式

描述字符形状的方式有很多种，图11-1给出了字符描述方式的分类。

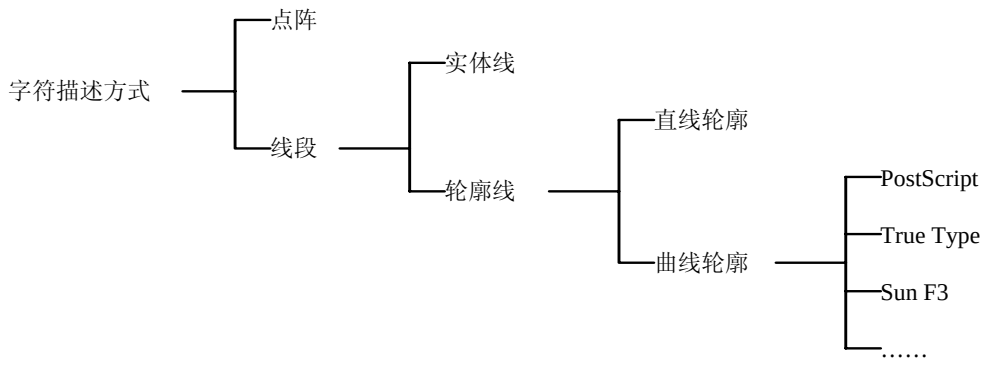

图11-1　字符描述方式分类

点阵描述是一种最基本的字符描述方式，具有结构简单、操作简便、输出速度快等优点，但有一个很大的缺点就是难以对字符进行高质量的缩放。较早期的汉字系统采用的都是点阵字库，在这些汉字系统中，为了能高质量地输出各种大小的字符，往往都对一种字体提供了多个不同大小的字库，常见的点阵汉字库的大小有16点阵、24点阵、32点阵、40点阵、48点阵等。英文点阵字库主要用作基本的字符输出，其常见的大小有16×8点阵和24×12点阵，英文点阵字库通常都是等宽字库。字符的点阵大小是点阵字库一个很重要的标识。

实体线描述方式就是用线段来描述出完整的字符形状，而非仅描述出字符形状的轮廓。与点阵描述方式相比，这种方式并不具有很明显的优点，主要适合于比例字体的描述。这种描述方式只在英文字库得到使用，且用得很少，但Borland C++所提供的英文字库采用的即是这种描述方式。

直线轮廓描述方式就是用直线段来描述字符形状的轮廓，采用这种描述方式的字库称为矢量字库，采用实体线描述方式的字库通常也称为矢量字库。矢量字库能够提供很好的字符缩放效果，但其字符缩放范围不能太大，能保证质量的缩放范围约为字符原大小的1/3至3倍，当缩得过小时，就会产生糊字现象，如果放得过大，就会看到较明显的折线痕迹。与各种曲线轮廓字库相比，矢量字库的缩放效果要差一些，但其复杂性也要小得多，

因此矢量字库得到了很广泛的使用。

曲线轮廓描述方式采用曲线来描述字符的轮廓，能提供极好的字符缩放效果。根据所采用的曲线类型的不同，这种描述方式又分为好几种，目前常见的有两种：PostScript和True Type，前者为WPS NT所采用，后者为Microsoft Windows所采用。为了得到更好的输出效果，在某些曲线轮廓字库中还含有一些提示信息(Hint)，还有一些则在字体驱动程序中加进了一些特殊的处理。曲线轮廓字库是目前最先进的一类字库，它已得到了越来越广泛的使用。

5. 字库结构

字库结构指字库中所有信息的数据结构。要想直接对一个字库进行操作，就必须掌握该字库的结构。一个字库的基本结构主要取决于字符描述方式及字符集，有时则还与字符的大小有关，一个字库的具体结构则是由字库的构造者确定的，在这方面并不存在统一的标准，不同的构造者所构造的相同类型的字库，其结构往往都是不同的，因此必须针对具体的构造者所提供的字库来讨论字库的操作方法，而不能一般性地针对字符描述方式、字符集或字符大小来讨论字库的操作。字库结构一般与字体无关，同一构造者所构造的不同字体的字库一般都具有相同的结构。

本书将介绍如下一些字库的操作技术，并实现相应的操作程序。

- 16×8点阵，24×12点阵ASCII字库；
- Borland C++10种字体的ASCII实体矢量字库；
- 本书所提供的1个ASCII轮廓矢量字库；
- 2.13汉字系统的16点阵、24点阵、32点阵、40点阵的图形汉字库及4种字体的基本汉字库；
- UCDOS 3.0的矢量图形汉字库及26种字体的矢量基本汉字库。

11.2 字库结构及操作方式

11.2.1 各类字库的基本结构

1. 点阵字库

点阵字库通常都是等宽体字库，在等宽体点阵字库中，每个字符的字形数据的长度都是相同的，如在一个16×16的点阵字库中，每个字符的字形数据的长度都为32个字节。在点阵字库中一般只存放有每个字符的字形数据，而不包含其它的任何信息，各个字符的字形数据按字符的机内码的大小，由小到大顺序存放在字库中，根据字符的机内码可以算出一个字符在字库的排列次序，将其乘以字形数据的长度即可得到该字符的字形数据在字库中的地址，由此地址即可读出该字符的字形数据。

在一个点阵字库中，每个字符的字形数据都对应于一个固定大小的二值图象，字形数据中的每个二进制位对应于图象上的一个象素点，当字形数据中的某位为1时，该位所对应的象素点在输出时就应置为指定的字符颜色，否则就保持原色不变。

2. 矢量字库

在矢量字库中，各个字符的字形数据的长度是不等的，这时就无法根据字符的排列次序来直接求得字形数据的地址，因而在矢量字库中都包含有一个指针数据区，在该数据区

内存放着每个字符的字形数据在字库中的地址，每个字符的指针数据也是按字符的机内码由小到大依次存放在指针数据区内，且每个字符的指针数据的长度都是相同的，这样，就可根据字符的机内码首先算出其指针数据的存放位置，然后在该位置读出字形数据的地址，再根据该地址读出字形数据。

在矢量字库中，一个字符的字形数据分为若干个笔划数据，每个笔划数据包含有若干个点，将笔划数据中的每个点顺序用直线相连，即可得到字符的一个笔划，多个笔划即构成一个字符。在轮廓矢量字库中，所给出的只是笔划的边框，需要由程序来负责对每个笔划进行填充，在实体矢量字库中，所给出的完整的笔划实体，程序不需要对其作进一步的处理。

ASCII矢量字库一般都是比例体字库，在这类字库中还需要包含有字符的宽度数据。每个字符的宽度数据都是等长的，因而可以根据机内码算出其存放位置，然后读取它。

3. 地址计算

字库中的那些等长数据都需要根据字符的机内码来计算其在字库中的存放位置，具体的计算方式与字符集有关，下面分别介绍。

(1)ASCII字库

计算公式如下：

$$Addr=Off+Size*(Code-FC) \tag{11-1}$$

Code——某个字符的ASCII码

FC——字库中所包含的第一个字符的ASCII码

Size——所要读的某个数据项对一个字符的长度

Off——该数据项的数据区在字库中的开始地址

Addr——该字符的该项数据的地址

对一定字库的一定数据项而言，FC、Size、Off都是常数。

(2)国标汉字库

计算公式如下：

$$Addr=Off+Size*[(Code1-0A1h)*94+Code2-0A1h] \tag{11-2}$$

Code1——某个汉字机内码的高字节

Code2——该汉字机内码的低字节

该计算公式也适合于图形汉字库。

(3)基本汉字库

计算公式如下：

$$Addr=Off+Size*[(Code1-0B0h)*94+Code2-0A1h] \tag{11-3}$$

11.2.2　ASCII点阵字库

1. 16×8点阵ASCII字库

这是一个等宽体字库，每个字符的字形数据的长度为16个字节，字形数据地址的计算公式为：Addr=Code*16。该字库包含了全部256个ASCII字符，字库文件的长度为4K，字库文件名为：ASC16。

16字节字形数据对应于一个16×8点阵的图象，两者间的映射关系为：字形数据中的每个字节顺序对应于字形图象中的每行象素点，在每个字节中，位7对应着最左边的象素

点，位0对应着最右边的象素点。

2. 24×12点阵ASCII字库

它是等宽体字库，每个字符的字形数据的长度为48个字节，字形数据地址的计算公式为：Addr=Code*48。其包含了全部ASCII字符，字库文件的长度为12K，字库文件名为：ASC24。

字形数据与字形图象之间的映射关系为：字形数据中的每两个字节顺序对应于字形图象中的每行象素点，在每一行的两个字节中，前12位依次对应于每行的12个象素点，后4位没有使用。

11.2.3 Borland C++的ASCII矢量字库

Borland C++的ASCII矢量字库是一种实体矢量字库，以下简称BC矢量字库。在Borland C++ 3.0及其以上的版本中，提供了10种字体的10个矢量字库，在其以下的版本中则只提供了4种，所有字体都是比例字体。这些字库都没有包含全部的ASCII字符，各个字库所包含的字符个数及起始字符存在着差别，各个字库中字符高度也是不同的，表11-1列出了这些字库的有关情况，所有这些字库都具有完全相同的结构。

表11-1 Borland C++的矢量字库

字库文件名	字 体	字符个数	起始字符	字符高度	文件长度/Byte
TRIP.CHR	Triplex	223	20h	32	16 677
LITT.CHR	Small	223	20h	10	5 131
SANS.CHR	Sans Serif	254	01h	33	13 596
GOTH.CHR	Gothic	223	20h	33	18 063
SCRI.CHR	Script	223	20h	38	10 987
SIMP.CHR	Simplex	223	20h	36	8 437
TSCR.CHR	Triplex Scr	223	20h	32	17 355
LCOM.CHR	Complex	223	20h	36	12 083
EURO.CHR	European	223	20h	56	8 439
BOLD.CHR	Bold	224	20h	61	14 670

在Borland C++ 3.0以下的版本中，只提供了表11-1中的前4个字库。

字库包含4个数据区：总体信息数据区、指针数据区、字符宽度数据区、字形数据区。

(1)总体信息数据区

处于字库的最前面，位置为：00H～8FH，包含了字库的一些总体信息，这些信息与具体的字符无关，所包含的数据项及位置如下：

00H～58H 一些文字提示信息

5BH～5EH 字库文件名

5FH～60H 字形数据区的长度

81H～82H 字库中所包含的字符个数，记字符个数为n

84H　　字库中第1个字符的ASCII码
85H～86H　字形数据区相对于80H的偏移
88H　　字符在基线上的高度
8AH　　字符延伸到基线下的高度，为一负数

(2)指针数据区

该数据区存放着每个字符的字形数据相对于字形数据区首的偏移，字形数据区首的位置从85H～86H处获得。各字符的指针数据按字符的ASCII码值由小到大顺序存放，每个字符的指针数据的长度为2个字节。该数据区开始于90H，长度为2n。

(3)字符宽度数据区

该数据区存放着每个字符的宽度值，数据按字符的ASCII码由小到大顺序存放，1个字符占1个字节。该数据区开始于90H+2n，长度为n。

(4)字形数据区

该数据区存放着各字符的字形数据。该数据区的开始位置存放于85H～86H。由于各字符的字形数据是不等长的，因此各字符的字形数据的位置不能直接算出，必须从指针数据区获取。

每个字符的字形数据由若干个字组成，每个字表示一个点，其第1字节和第2字节的低7位分别为点的x坐标值和y坐标值，两个坐标值都有符号数，取值范围为-63～64。所采用的坐标系为实际坐标系，即向上为y的正方向，坐标原点处于字符的左下角。两个字节的最高位为两个控制码，分别记为cc1和cc2，程序需根据这两个控制码来在各点间画直线，这两个控制码的含义如下：

若cc1=1、cc2=0，表示该点为某个笔划的第一个点；
若cc1=1、cc2=1，表示应在上一个点至该点间连一条直线；
若cc1=0、cc2=0，表示当前字符的字形描述结束。

11.2.4　一种ASCII轮廓矢量字库

Borland的矢量字库是实体矢量字库，这种字库不适于作放大输出，放大会在字符笔划中形成一些空洞，因此本书专门提供了一个ASCII轮廓矢量字库，其能较好地用作大字符的输出。该字库只包含了ASCII码值为32至127个的96个字符，字符的高度为96，字库文件的长度为7376字节，字库文件名为：ASCSL。

该字库包含3个数据区：指针数据区、字符宽度数据区、字形数据区。

(1)指针数据区

处于字库的最前面，位置为：00H～BFH。该数据区存放着每个字符的字形数据的地址，每个字符的地址数据的长度为2个字节。某个字符的指针数据位置的计算公式为：Addr=2*(Code-20H)。

(2)字符宽度数据区

位置为：A0H～11FH。存放着每个字符的宽度值，每个字符占一个字节，某个字符的宽度数据位置的计算公式为：Addr=A0H+(Code-20H)。

(3)字形数据区

开始于120H，结束于文件尾。存放着每个字符的字形数据，各字符的字形数据的存放位置需从指针数据区中获得。

每个字符的字形数据由若干段笔划数据所组成。每个笔划实际上就是一个多边形，每段笔划数据的第1个字节为多边形的顶点数，以后依次存放着各顶点的x、y坐标值，各项数据都是单字节的无符号数。若某段笔划数据的第1个字节为0，则表明当前字符的笔划数据结束。所采用的是屏幕坐标系，即向下为y的正方向，坐标原点为字符的左上角。

11.2.5 2.13的点阵汉字库

2.13汉字系统提供了4种大小的16个点阵汉字库，这些字库的有关情况列于表11-2。

表11-2 2.13点阵汉字库

字库文件名	字符点阵（宽×高）	字符集	字体	字形数据长度/Byte	文件长度/Byte
HZK16	16×16	国标字符集	宋	32	261 696
HZK24T	24×24	图形字符集		72	101 520
HZK24S	24×24	基本字符集	宋	72	487 296
HZK24H	24×24	基本字符集	黑	72	487 296
HZK24K	24×24	基本字符集	楷	72	487 296
HZK24F	24×24	基本字符集	仿宋	72	487 296
HZK32T	32×32	图形字符集		128	120 320
HZK32S	32×32	国标字符集	宋	128	866 304
HZK32H	32×32	国标字符集	黑	128	866 304
HZK32K	32×32	国标字符集	楷	128	866 304
HZK32F	32×32	国标字符集	仿宋	128	866 304
HZK40T	36×40	图形字符集		180	169 200
HZK40S	36×40	国标字符集	宋	180	1 218 240
HZK40H	36×40	国标字符集	黑	180	1 218 240
HZK40K	36×40	国标字符集	楷	180	1 218 240
HZK40F	36×40	国标字符集	仿宋	180	1 218 240

点阵汉字库的结构决定于字符点阵大小和字符集。字符集决定了字形数据地址的计算公式，具体见式11-2和式11-3，对点阵字库来说，式中的参数Off总为0，参数Size则为表11-2中所列出的字形数据长度，字形数据的长度也可根据字符点阵大小按下式直接计算：字形数据长度(Byte)＝字符的水平象素数×垂直象素数／8。

字形数据与字形图象间的映射关系则与字符点阵的大小有关，下面分别介绍各种点阵大小的字库中字形数据与字形图形间的映射关系。

1. 16点阵

32字节的字形数据对应于一个16×16点阵的二值图象。1个字节对应于一行象素中连续的8个象素点，其中位7对应于8个象素点中最左边的1个，位0对应于最右边的1个。1行

象素由连续的2个字节来表示，前面的字节对应于左边的8个象素点。按由上到下的顺序，每2个字节依次对应于1行象素点。

2. 24点阵

72字节的字形数据对应于一个24×24点阵的二值图象。1个字节对应于一列象素中连续的8个象素点，其中位7对应于8个象素点中最上面的1个，位0对应于最下面的1个，32点阵、40点阵的字库也都采用的是这种纵向对应方式。连续的3个字节按从上到下的顺序对应于1列象素。按从左到右的顺序，每3个字节依次对应于各列象素。

3. 32点阵

128字节的字形数据对应于一个32×32点阵的二值图象。每个字节采用的都是纵向对应方式。图象分为上下两部分：第一部分的高度为8个象素点，其对应于前32个字节，每个字节对应于1列象素，各字节从左到右依次对应于各列象素；第二部分图象的高度为24个象素点，其对应于后96个字节，连续的3个象素点从上到下对应于1列象素，每3个字节从左到右依次对应于各列象素。

4. 40点阵

180字节的字形数据对应于一个36×40点阵的二值图象。每个字节采用的都是纵向对应方式。图象分为上下两部分：第一部分的高度为16个象素点，其对应于前72个字节，连续的2个字节从上到下对应于1列象素，每2个字节从左到右依次对应于各列象素；第二部分的高度为24个象素点，其对应于后108个字节，连续的3个字节从上到下对应于1列象素，每3个字节从左到右依次对应于各列象素。

其它汉字系统所提供的点阵汉字库的结构绝大部分都与2.13的相同。

11.2.6　UCDOS的矢量汉字库

1. 基本情况

UCDOS 3.0/3.1汉字系统提供了26种字体的矢量基本汉字库和一个矢量图形汉字库，这些字库都是等宽体的轮廓矢量字库，表11-3列出了其中5个基本字库的有关情况。

表11-3　UCDOS五个基本矢量字库的有关情况

字库文件名	字符集	字体	字符大小/Byte	文件长度/Byte
HZKSLT	图形字符集		128	131 232
HZKSLSTJ	基本字符集	宋体	96	1 243 953
HZKSLHTJ	基本字符集	黑体	96	1 199 709
HZKSLKTJ	基本字符集	楷体	96	1 402 605
HZKSLFSJ	基本字符集	仿宋	96	1 038 009

2. 字形数据的读取

UCDOS的矢量字库包含两个数据区：指针数据区和字形数据区。指针数据区位于字库的最前面，其中不仅存有各字符字形数据的地址,而且存有各字符字形数据的长度，每个字符在指针数据区中占6个字节，其中前4个字节以长整型数的格式存放着字形数据在字库中的绝对地址，后两个字节以整型数的格式存放着字形数据的长度，各字符的指针数据的存放位置可按如下公式算出：

对图形汉字库：Addr=6*[(Code1-0A1h)*94+Code2-0A1h]

对基本汉字库：Addr=6*[(Code1-0B0h)*94+Code2-0A1h]

指针数据区之后即为字形数据区，由于指针数据区中存放的是各字符字形数据的绝对地址，因此字形数据区的开始位置是无关紧要的。

首先根据字符的机内码算出字符的指针数据的存放位置，从中读出字符的字形数据的地址和长度，由该地址及该长度即可读出字符的字形数据。

3. 字形数据的格式

一个字符的字形数据由若干个多边形数据段组成，每个多边形数据段中包含有该多边形的所有顶点的坐标值，除第1个顶点外其它顶点的坐标值都不是直接给出的，而是给出了相对于上一个顶点的偏移值，偏移值的表示方式有多种，一种偏移值表示方式往往连续用于相邻的多个顶点，那些具有相同偏移值表示方式的若干个相邻顶点，称为一个顶点数据段，表示第1个顶点的数据可视为一个特殊的顶点数据段，一个多边形数据段即由若干个顶点数据段所组成。每个顶点数据段的第1个字节为一个控制码，记为cc，其中的高4位记为cc1，低4位记为cc2，cc1表示了顶点数据段的类型或偏移值的形式，cc2或者用于表示顶点数据段所包含的顶点个数，或者用于表示顶点的值，由cc1和cc2可确定出顶点数据段的数据长度，各顶点数据段及多边形数据段连续存放，之间没有分隔标识，因此只能根据顶点数据段的长度来判别下一个顶点数据段的开始。所采用的是屏幕坐标系，坐标原点为字符的左上角。控制码的具体含义如下：

①若cc1≥12。为多边形的第1个顶点数据段，该数据段的长度总为2个字节(包含控制码字节)，记为Byte1和Byte2，其中Byte1=cc，这两个字节给出多边形第1个顶点的x，y坐标值，计算方式如下：x=(Byte1-12)*2+(Byte2>>7)，y=Byte2&0x7F。

②若cc1=11。该数据段的长度总为3个字节，其后两个字节Byte2和Byte3用于表示当前一个顶点相对于上一个顶点的坐标偏移值dx，dy，计算方式如下：

```
dx=Byte2          若Byte2<=0x80
dx=0x80-Byte2     若Byte2>0x80
dy=Byte3          若Byte3<=0x80
dy=0x80-Byte3     若Byte3>0x80
```

③若cc1=9。该数据段的长度总为2个字节，第1个字节的低4位cc2和第2个字节Byte2分别表示了dx和dy，计算方式如下：

```
dx=cc1            若cc1<=8
dx=8-cc1          若cc1>8
dy=Byte2          若Byte2<=0x80
dy=0x80-Byte2     若Byte2>0x80
```

④若cc1=8。该数据段的长度总为2个字节，第1个字节的低4位cc2和第2个字节Byte2分别表示了dy和dx，计算方式如下：

```
dy=cc1            若cc1<=8
dy=8-cc1          若cc1>8
dx=Byte2          若Byte2<=0x80
dx=0x80-Byte2     若Byte2>0x80
```

⑤若cc1=7。该数据段包含了若干个向右上方向偏移的顶点，cc2为所包含的顶点个

数，该数据段的长度为cc2+1个字节，第1个字节之后的每个字节分别对应于一个顶点，每个字节的高4位和低4位分别表示了一个顶点向右上方向的x和y偏移值，记某个字节的高4位为b1，低4位为b2，则有：dx=b1，dy=-b2。

⑥若cc1=6。类似于cc1=7，这时有：dx=-b1，dy=-b2。

⑦若cc1=5。类似于cc1=7，这时有：dx=-b1，dy=b2。

⑧若cc1=4。类似于cc1=7，这时有：dx=b1，　dy=b2。

⑨若cc1≤3。该数据段包含了若干个顶点，cc为所包含的顶点个数，该数据段的长度为cc+1个字节，第1个字节之后的每个字节分别对应于一个顶点，每个字节的高4位和低4位分别表示了一个顶点的x和y偏移值，记某个字节的高4位为b1，低4位为b2，则有：

dx=b1　　若b1<=8
dx=8-b1　　若b1>8
dy=b2　　若b2<=8
dy=8-b2　　若b2>8

4. 字形填充

轮廓矢量字库只提供了字符形状的边框，程序必须对字形边框进行填充才能完成有效的字符输出。每个字形边框实际上就是一个多边形，按多边形填充方法即可实现对一个字形边框的填充，但一个字符通常都包含有多个字形边框，如何对一个字符的多个边框进行填充则与字符轮廓的生成方式有关，从填充的角度考虑，字符轮廓的生成方式可有如下3种。

①对每个笔划构造一个边框，如，对一个“十”字，用两个字形边框来描述它，分别描述一横和一竖。在这种情况下，就需对一个字符的多个边框分别进行填充，如同时进行填充就会在笔划的交叉处产生空洞。

②对字形中每个连通的区域用一个边框或两个边框(有空洞时)来描述，如，用一个边框来描述“十”字，用两个边框来描述“口”字，一个描述外框，一个描述内框。在这种情况下，就需对一个字符的多个边框同时进行填充，否则当存在空洞时就会把空洞也填充掉，如对“口”字，分别填充其内、外框，就会将其内部的空洞填充掉。

③上述两种方式的混合，即在同一个字库中或在同一个字符内，同时采用了上述两种方式，如，一个“古”字，上面的“十”用第1种方式描述，下面的“口”用第2种方式描述。在这种情况下，填充就需按如下方式进行：如果两个或多个边框间存在着包含关系，则同时填充这些边框，如果一个边框不与其它边框有包含关系，则单独填充此边框。由于一个字符内的任何两个边框间都有可能存在包含关系，因而必须对各边框逐一进行组合判别，其计算量是比较大的。如果想免去大量的计算，那就最好采用同时填充的方式，这会在笔划交叉处产生空洞，但与糊字相比，其损害还是要小一些。

UCDOS的5个基本矢量字库中，宋、黑、仿宋3个字库属第1种情况，图形、楷体两个字库属第3种情况。

11.3　小 汉 字 库

即使是最小的汉字库其大小也有260K，最大的则达到了1.4M，不论是放在内存中还是放在硬盘上，它们所占用的空间都是很大的，而很多应用程序往往只是用到了整个国标字

符集中的少数几十个或几百个字符，如果将一个程序所要用到的那些汉字字符从字库中抽取出来，构造一个专用的小汉字库供程序使用，那就能大大节省字库所占用的空间。这一节介绍如何构造专用的小汉字库，并实现一个构造小汉字库的实用程序。ASCII字库本身都是很小的，因而没有必要构造ASCII字符的小字库。

11.3.1 小字库的结构

并不存在一种标准的或常用的小字库结构，这里所介绍的小字库的结构是由本书所确定的。

1. 基本结构

在小字库中，显然再不能按字符的机内码直接算出字符的字形数据或指针数据在字库中的存放位置，因而在原字库原有数据区的基础上，在小字库中增加一个字符数记录和一个索引数据区，字符数记录放在字库的最前面，其为2个字节，存放着字库中所包含的字符的个数，记为n。字符数记录之后即为索引数据区，在该数据区内，按由小到大的顺序连续存放着字库所包含的n个字符的机内码，每个字符机内码的长度为2个字节，因此索引数据区的长度为2n个字节。索引区之后的数据结构则与字库的类型有关，下面分别讨论2.13的各种点阵字库所对应的小字库及UCDOS的矢量字库所对应的小字库的具体字库结构。

2. 小点阵字库的结构及读取方式

索引数据区之后即为字形数据区，字形数据区内存放着n+1个字符的字形数据，其中第1个字形数据为一个缺省字形数据，那些不包含在字库中的字符或者说在索引区中找不到的字符都将使用此缺省字形数据，缺省字形数据之后即为包含在字库中的n个字符的字形数据，这些字形数据按字符的机内码由小到大顺序存放。字形数据的长度与原字库相同。

当要从字库中读取某个字符的字形数据时，首先在索引区查找该字符的机内码，若找到了，设其机内码位于索引区的第i个字(i=1～n)，若未找到，则有i=0。设每个字形数据的长度为Size，则可按下式计算得到该字符的字形数据在字库中的存放位置Addr。

$$Addr=2(n+1)+Size*i$$

所读出的字形数据与字形图象间的映射关系与原字库相同。

3. 小矢量字库的结构及读取方式

索引数据区之后为指针数据区，该数据区内存放着n+1个字符的指针数据，其中第1个指针数据指向一个缺省的字形数据，之后即为字库所包含的n个字符的指针数据，这些指针数据按字符的机内码由小到大顺序存放。指针数据的长度为6个字节，前4个字节为字形数据在字库中的地址，后2个字节为字形数据的长度，这与原字库相同。指针数据区之后为字形数据区，该数据区存放着n+1个字符的字形数据，其中第1个字形数据为一个缺省的字形数据，其后则为字库所包含的n个字符的字形数据。

当要从字库中读取某个字符的字形数据时，首先在索引区查找该字符的机内码，若找到了，设其机内码位于索引区的第i个字(i=1～n)，若未找到，则有i=0。按下式计算出该字符的指针数据在字库中的存放位置Addr。

$$Addr=2(n+1)+6*i$$

从Addr处读出该字符的字形数据在字库中的地址及长度，再由此读出该字符的字形数

据。所读出的字形数据的格式及对字形轮廓的填充方式与原字库相同。

11.3.2　小字库构造程序

这里实现了一个构造小字库的实用程序。程序的基本功能为：从一个文本源文件中搜索出其所包含的所有汉字字符，再从一个字库中取出这些字符的字形数据，然后将数据存入一个指定小字库中，如该小字库已经存在，将保留其原有字符并将新的字符补充进去。在24点阵、32点阵、40点阵的字库中，图形汉字字符和基本汉字字符是分开的，在小字库中将合并它们，在UCDOS的矢量字库中，图形字符和汉字字符的大小是不同的，因此在小字库中没有合并它们。

在运行该程序时，需要从命令行输入如下一些参数：字库类型，原字库文件名，文本源文件名，小字库文件名。如果字库类型是24、32、40点阵的字库，则原字库文件名必须是两个，一个为基本汉字库的文件名，一个为图形汉字库的文件名。从命令行输入的参数被自动保存在由<DOS.H>所定义的一个全局变量*_argv[]中，所输入的命令行参数的个数则保存在全局变量_argc中。

应用程序11-1给出了一个完整的小字库构造程序，在程序中，针对每种具体的字库分别定义了一个类，构造各种小字库的功能都包含在相应的类中，这些类都派生于一个基类，在该基类中实现了那些构造小字库的公用功能。

应用程序11-1　XHZK.CPP

```
--------------------------------------------------------------------------------
#include <stdio.h>
#include <dos.h>
#include <alloc.h>
#include <string.h>

union CTOI {  //该联合用于将2字节的汉字机内码转换为一个整型数
    unsigned ii;
    unsigned char cc[2];
    } _ctoi;

/*  小字库构造功能的基类  */
class CHZS {
public:
    virtual void run(void)=0;  //小字库构造功能的接口函数，其需针对具体的字库实现
protected:
    int MAXN;  //小字库最多能包含的字符数
    long Isize;  //字形数据或指针数据的长度
    int coden,orgn;
    unsigned *code;  //存放小字库中所包含字符的机内码

    int ifrun(int n);  //判别程序的输入参数是否正确
    void savehead(char *filename);  //存小字库的索引数据
    void getcode(char *filezj,char *filed);  //从文本源文件中获得汉字字符
    void codesort();  //对小字库所包含的字符的机内码进行排序
    int findcode(unsigned ii);  //判别从文本源文件中所找到的一个字符是否是重复的
    virtual long getoffset(unsigned char *cp);
```

```
      //获得一个字符的字形数据或指针数据在原字库中的位置
    virtual int ifin(unsigned char c);  //判别一个字节是否可能是机内码的首字节
    } ;

/*  构造16点阵小字库的类  */
class CHZ16S : public CHZS
    {
public:
    CHZ16S() { MAXN=1000; Isize=32; }
    void run(void);
protected:
    long getoffset(unsigned char *cp);
    } ;

/*  构造24点阵小字库的类  */
class CHZ24S : public CHZS
    {
public:
    CHZ24S() { MAXN=900; Isize=72; }
    void run(void);
    } ;

/*  构造32点阵小字库的类  */
class CHZ32S : public CHZ24S
    {
public:
    CHZ32S() { MAXN=500; Isize=128; }
    } ;

/*  构造40点阵小字库的类  */
class CHZ40S : public CHZ24S
    {
public:
    CHZ40S() { MAXN=350; Isize=180; }
    } ;

/*  构造UCDOS矢量汉字小字库的类  */
class CHZUCS : public CHZS
    {
public:
    CHZUCS() { MAXN=200; Isize=6; }
    void run(void);
protected :
    int ifin(unsigned char c);
    } ;

/*  构造UCDOS矢量图形小字库的类  */
class CTXUCS : public CHZUCS
    {
```

```
protected :
    int ifin(unsigned char c);
    } ;

/*************************************************************************
功能：判别程序输入参数的个数是否满足要求及所指定的有关文件是否存在
输入参数：n＝运行程序时必须从命令行输入的参数的个数
返回值：int＝所给出的参数是否满足程序运行的要求
            1  程序可以运行
            0  程序不能运行
*************************************************************************/
int CHZS::ifrun(int n)
{
int i;
FILE *fp;

if( _argc<=n )  //所输入的参数的个数小于所要求的个数
    {
    printf("\n Parameter too few.\n");
    return(0);
    }

/*  以下循环，判别原字库文件及文本源文件是否存在  */
for(i=2;i<n;i++)
    {
    fp=fopen(_argv[i],"rb");
    if(fp==NULL)
        {
        printf("\n The file %s no exists.\n",_argv[i]);
        return(0);
        }
    fclose(fp);
    }
return(1);
}

/*************************************************************************
功能：写小字库的字符数记录及索引数据
输入参数：filename＝小字库文件名
返回值：无
*************************************************************************/
void CHZS::savehead(char *filename)
{
FILE *fp;
fp=fopen(filename,"wb");
fwrite(&coden,2,1,fp);
fwrite(code,coden,2,fp);
fclose(fp);
}
```

```
/*************************************************************************
功能：从文本源文件中读出新的汉字字符
输入参数：filezj＝文本源文件名
          filed＝小字库文件名
返回值：无
*************************************************************************/
void CHZS::getcode(char *filezj,char *filed)
{
FILE *fp;
unsigned char c1,c2;

coden=orgn=0;
fp=fopen(filed,"rb");  //试着打开小字库
if(fp!=NULL)  //如果小字库已经存在，读出小字库中已有的字符，放在code中
    {
    fread(&orgn,2,1,fp);
    fread(code,2,orgn,fp);
    fclose(fp);
    }
/*  以下，从文本源文件中读取新的汉字字符，补充到code中  */
coden=orgn;
fp=fopen(filezj,"rb");
while(coden<MAXN) {
    if(fread(&c1,1,1,fp)<1)  //文件结束
        breaK;
    if( ifin(c1) )  //所读出的字节可能为当前字符集的机内码的第1个字节
        {
        if(fread(&c2,1,1,fp)<1)  //读取第2个字节，并判别文件是否结束
            breaK;
        if(c2>=0xa1)  //第2个字节满足要求，确认这两个字节为一个汉字字符
            {
            _ctoi.cc[0]=c1;
            _ctoi.cc[1]=c2;
            if( findcode(_ctoi.ii)==0 )  //这个字符在code中不存在，将其放到code中
                {
                code[coden]=_ctoi.ii;
                coden++;
                }
            }
        }
    }
fclose(fp);
}

/*************************************************************************
功能：判别一个字符是否已经存在于CHZS::code中
输入参数：ii＝所要判别字符的机内码
返回值：int＝1  已经存在
            0  不存在
```

```
**************************************************************************/

int CHZS::findcode(unsigned ii)
{
if(coden==0)
    return(0);
int beg,end,i,K;

beg=0;
end=coden-1;
i=coden/2;
K=0;
while(1) {
    if(end-beg<=1)
        {
        K++;
        if(K>2)
            return(0);
        }
    if(ii==code[i])
        return(1);
    else if(ii>code[i])
        {
        beg=i;
        i=(beg+end+1)/2;
        }
    else
        {
        end=i;
        i=(beg+end)/2;
        }
    }
}

/*************************************************************************
功能：对CHZS::code中的字符进行排序
输入参数：无。直接使用CHZS::coden和CHZS::code
返回值：无。排序结果仍放在CHZS::code中
*************************************************************************/
void CHZS::codesort()
{
int i,f;
unsigned K;

while(1) {
    f=0;
    for(i=0;i<coden-1;i++)
        if(code[i]>code[i+1])
            {
```

```
            f=1;
            K=code[i];
            code[i]=code[i+1];
            code[i+1]=K;
            }
        if(f==0)
            breaK;
    }
}

/*************************************************************************
功能：计算某个字符的字形数据或指针数据在原字库中的存放位置
输入参数：cp＝2个字节的机内码
返回值：long＝数据的存放位置
    这里所实现的这一函数仅分别对图形汉字集或基本汉字集有效，对国标汉字集无效，因而对16点阵的汉字库，该函数需重写。
*************************************************************************/
long CHZS::getoffset(unsigned char *cp)
{
long off;
if(cp[0]<0xb0)  //为图形汉字库
    off=(long)Isize*( ((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L );
else  //为基本汉字库
    off=(long)Isize*( ((long)cp[0]-0xb0L)*94L+(long)cp[1]-0xa1L );
return(off);
}

/*************************************************************************
功能：判别一个字节是否为汉字机内码的第1个字节
输入参数：c＝所要判别的字节
返回值：int＝1：是；0：否
    这里实现的这一函数只对国标汉字集有效，对基本汉字集和图形汉字集无效。
*************************************************************************/
int CHZS::ifin(unsigned char c)
{
return (c>=0xa1);
}

/*  构造16点阵小字库功能的接口函数  */
void CHZ16S::run()
{
if( ifrun(4)==0 )  //输入参数不满足要求
    return;

unsigned i;
void *buf;
FILE *fps,*fpd;
long off;
```

```
code=(unsigned *)calloc(MAXN,2);
getcode(_argv[3],_argv[4]);  //获得小字库所包含的字符
codesort();  //排序
savehead(_argv[4]);  //写小字库的字符数记录及索引数据

fps=fopen(_argv[2],"rb");  //打开原16点阵字库
fpd=fopen(_argv[4],"ab");  //打开小字库
buf=calloc(Isize,1);
fwrite(buf,Isize,1,fpd);  //向小字库写入一个缺省的字形数据
for(i=0;i<coden;i++)
    {
    _ctoi.ii=code[i];
    fseeK(fps,getoffset(_ctoi.cc),0);
    fread(buf,Isize,1,fps);  //读原字库中的字形数据
    fwrite(buf,Isize,1,fpd);  //写到小字库中
    }
fclose(fps);
fclose(fpd);
free(buf);
}

/*  计算某个字符的字形数据在16点阵字库中的位置  */
long CHZ16S::getoffset(unsigned char *cp)
{
long off;
off=(long)Isize*( ((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L );
return(off);
}

/*  构造24、32、40点阵小字库功能的接口函数。其要完成图形汉字集与基本汉字集的合并  */
void CHZ24S::run()
{
if( ifrun(5)==0 )  //输入参数不满足要求
    return;

unsigned i;
void *buf;
FILE *fps1,*fps2,*fpd;
long off;

code=(unsigned *)calloc(MAXN,2);
getcode(_argv[4],_argv[5]);
codesort();
savehead(_argv[5]);

fps1=fopen(_argv[2],"rb");  //打开原基本汉字库
fps2=fopen(_argv[3],"rb");  //打开原图形汉字库
fpd=fopen(_argv[5],"ab");  //打开小字库
buf=calloc(Isize,1);
```

```
fwrite(buf,Isize,1,fpd);
for(i=0;i<coden;i++)
    {
    _ctoi.ii=code[i];
    off=getoffset(_ctoi.cc);
    if(_ctoi.cc[0]<0xb0)  //如果是图形汉字，从图形汉字库中读取其数据
        {
        fseeK(fps2,off,0);
        fread(buf,Isize,1,fps2);
        }
    else  //如果是基本汉字，从基本汉字库中读取其字形数据
        {
        fseeK(fps1,off,0);
        fread(buf,Isize,1,fps1);
        }
    fwrite(buf,Isize,1,fpd);
    }
fclose(fps1);
fclose(fps2);
fclose(fpd);
free(buf);
}

/*  构造UCDOS矢量小字库功能的接口函数  */
void CHZUCS::run()
{
if( ifrun(4)==0 )
    return;

unsigned i;
void *buf;
FILE *fps,*fpd;
long off,bhoff;
int leng;

code=(unsigned *)calloc(MAXN,2);
getcode(_argv[3],_argv[4]);
codesort();
savehead(_argv[4]);

fps=fopen(_argv[2],"rb");
fpd=fopen(_argv[4],"ab");
bhoff=2L+coden*2L+(coden+1)*6L;  //计算字形数据区的开始位置
leng=1;
fwrite(&bhoff,4,1,fpd);  //写入缺省字形数据的地址
fwrite(&leng,2,1,fpd);  //写入缺省字形数据的长度
bhoff += (long)leng;
/*  以下循环，向小字库中写各字符的指针数据  */
for(i=0;i<coden;i++)
```

```
    {
    _ctoi.ii=code[i];
    off=getoffset(_ctoi.cc)+4L;
    fseeK(fps,off,0);
    fread(&leng,2,1,fps);
    fwrite(&bhoff,4,1,fpd);
    fwrite(&leng,2,1,fpd);
    bhoff += (long)leng;
    }
buf=calloc(1024,1);
fwrite(buf,1,1,fpd);  //写入缺省的字形数据
/*  以下循环，向小字库中写各字符的字形数据  */
for(i=0;i<coden;i++)
    {
    _ctoi.ii=code[i];
    fseeK(fps,getoffset(_ctoi.cc),0);
    fread(&bhoff,4,1,fps);
    fread(&leng,2,1,fps);
    fseeK(fps,bhoff,0);
    fread(buf,leng,1,fps);
    fwrite(buf,leng,1,fpd);
    }
fclose(fps);
fclose(fpd);
free(buf);
}

/*  判别一个字节是否为基本汉字集中字符机内码的第1个字节  */
int CHZUCS::ifin(unsigned char c)
{
return( c>=0xb0 );
}

/*  判别一个字节是否为图形汉字集中字符机内码的第1个字节  */
int CTXUCS::ifin(unsigned char c)
{
return( c>=0xa1 && c<0xb0 );
}

/*  主函数。其根据由命令行所输入的存放在_argv[1]中的字库类型标志，动态定义与字库类型相
对应的一个类对象，并通过该对象调用相应的构造小字库的接口函数  */
void main()
{
CHZS *A;
if(stricmp(_argv[1],"HZ16")==0)
    A=new CHZ16S;
else if(stricmp(_argv[1],"HZ24")==0)
    A=new CHZ24S;
else if(stricmp(_argv[1],"HZ32")==0)
```

```
    A=new CHZ32S;
else if(stricmp(_argv[1],"HZ40")==0)
    A=new CHZ40S;
else if(stricmp(_argv[1],"HZUC")==0)
    A=new CHZUCS;
else if(stricmp(_argv[1],"TXUC")==0)
    A=new CTXUCS;
else
    {
    printf("\nParameter error\n");
    return;
    }
A->run();
delete A;
}
```

对上面的源程序进行编译、连接即可得到一个构造小字库的执行程序CHZS.EXE。

11.3.3 小字库构造程序使用说明

一个应用程序能使用小字库的基本前提是：该程序所要用到的汉字字符是固定的，能预先明确的，即在程序的运行过程中不会出现用户直接或间接输入新的汉字字符的情况。

这里介绍上面所实现的小字库构造程序CHZS.EXE的使用方式。

1.使用格式

XHZK type source_hzK [source_hzK1] text small_hzK

type 字库类型。其只能为如下几个字符串之一

HZ16 16点阵汉字库

HZ24 24点阵图形汉字库、4种字体的基本汉字库

HZ32 32点阵图形汉字库、4种字体的基本汉字库

HZ40 36×40点阵图形汉字库、4种字体的基本汉字库

HZUC UCDOS 26种字体的矢量基本汉字库

HZTX UCDOS矢量图形汉字库

所指定的字库类型必须与所给定的原字库相匹配。

source_hzK 原字库文件名

source_hzK1 第2个原字库的文件名。在字库类型为HZ24、HZ32或HZ40时，必须输入该参数，且此时，szK必须为基本汉字库的文件名，szK1必须为相应的图形汉字库的文件名。当字库类型不为这三种时，必须省略该参数。

text 包含有所需汉字字符的文本文件名

small_zK 小字库文件名

使用示例：

```
XHZK HZ16 hzK16 zj.txt hzK16.sml
XHZK HZUC c:\ucdos\hzKslstj zj.txt slst.sml
XHZK hz24 hzK24K hzK24t zj.txt hzK24K.sml
```

2.应用举例

假设有一个用C++编写的应用程序，该程序需要显示一些固定的汉字，所需显示的汉字字符全部包含在三个C++的源程序中，这三个源程序为：holt1.cpp、holt2.cpp、holt3.cpp。大部分字符需要以24点阵的楷体进行显示，有少数几个字符则需用UCDOS的黑体矢量字库作放大显示。下面介绍为这个应用程序构造小字库的方法。

首先，获得24点阵的楷体字库HZK24K，24点阵的图形字库HZK24T及UCDOS的黑体矢量字库HZKSLHTJ，将它们放到了HZKS.EXE所在的目录中。

用如下三行命令建立24点阵的小字库：

```
XHZK HZ24 hzK24K hzK24t holt1.cpp hzK24.sml
XHZK HZ24 hzK24K hzK24t holt2.cpp hzK24.sml
XHZK HZ24 hzK24K hzK24t holt3.cpp hzK24.sml
```

将那几个需要放大显示的汉字输入到一个文本文件中，设该文本文件为：lc.txt，然后用如下命令建立小矢量字库：

```
XHZK HZUC hzKslhtj lc.txt hzKsl.sml
```

11.4　字符显示程序设计

11.4.1　程序的功能

所要实现的字符显示程序将具备如下一些功能：

- 支持本章前面所介绍的各种字库的字符显示，所支持的字库包括：点阵ASCII字库、Borland的10种ASCII实体矢量字库、本书所提供的ASCII轮廓矢量字库、各种大小、字体的点阵汉字库及其小字库、UCDOS的各种字体的矢量汉字库及其小字库。
- 能在基础图形显示程序所支持的18种图形模式下进行字符显示，当基础图形显示程序所支持的图形模式得到扩充时，能自然对那些新的图形模式提供同样的支持。
- 能在常规内存、扩展内存、硬盘三种存储区域装载所要操作的字库，并能进行存储区域的容量检测和存储区域的自动转换。
- 允许应用程序同时使用多个字库，并允许在一个字符串的输出中同时使用分属于ASCII字符集、图形汉字字符集、基本汉字字符集的三个字库。

11.4.2　程序结构

程序所要支持的各个字库，它们的操作方式有的是相同的，有的则是不同的，字库在操作方式上的差别来自于四个方面：字符集；字形数据及宽度数据的读取方式；字形数据的格式；填充方式。如果两个字库在这四个方面的任一方面存在差别，它们的操作方式也就必然存在差别，如果两个字库在这四个方面都是相同的，它们的操作方式也就是相同的。根据各个字库在这四方面的情况，可从操作方式上，将程序所要支持的各个字库分为21种。在操作方式上，这21种字库之间都存在着不同程度的差别，但它们也存在着很多相同的地方，这一特点与图形显示程序非常类似，因而这里的字符显示程序也很适合于用C++来编写。

在程序中，针对这21种字库，每一种定义一个字符显示类，然后对各个类中相同的地方进行归纳，经过几个层次的归纳，形成了一个如图11-2所示的字符显示类体系，该类体系能以很小的代码冗余实现对各种字库字符显示的支持。图中带虚框的类为抽象类，它们不能实例化，其余的21个类都为可实例化的类，在应用程序中，这些类的一个对象即对应于一个具体的字库，对该字库的所有字符显示操作都包含在该对象之中。

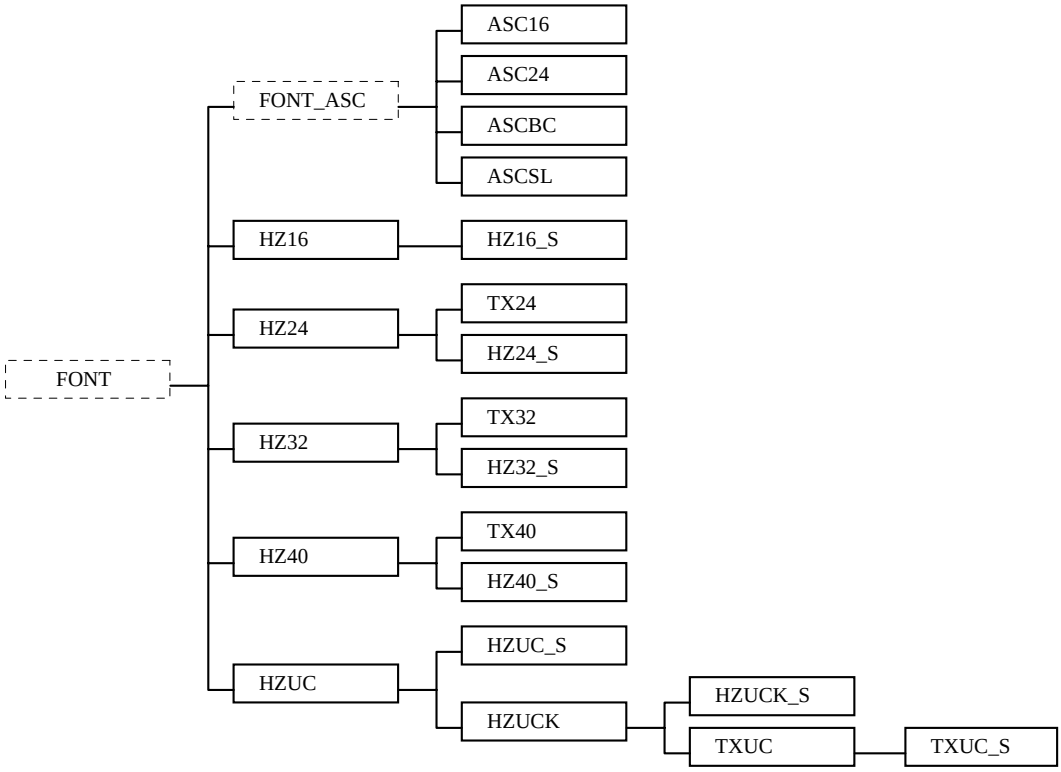

图11-2 字符显示类体系

11.4.3 类说明

这里给出字符显示类体系中各个类的说明。

系统程序11-1 VGAFONT.H

```
--------------------------------------------------------------------------------
class VGABASE;
enum font_type {  //该枚举提供了字库类型的助记符
    ASCfont = 0,
    TXfont  = 1,
    HZfont  = 2 };

union FONThandle {  //该联合定义了字库存放在各种不同区域的通用指针
    void *mem;  //字库装于常规内存时的指针
    class XMS *xms;  //字库装于扩展内存时的指针
    FILE *file;  //字库装于硬盘时的指针
    } ;

/*  字符显示基类  */
```

```
class FONT {
public:
    int OK;  //字库是否可用的标志
    int where;  //字库存放位置
    long leng;  //字库长度
    union FONThandle handle;  //字库指针

    FONT(char *filename,int whe,VGABASE *v);
    ~FONT();
    void setvga(VGABASE *v);  //确定显示模式
    void join(FONT *ftp1,FONT *ftp2=0);  //连接进另外两个字符集的字库
    void cut(FONT *ftp);  //切断字库间的连接
    void setsize(int hi,int wi,int xxi=0);  //设置字符大小及间隔，包括所连接进的字库
    void setsizeOnly(int hi,int wi,int xxi=0);  //设置字符大小及间隔，仅对本字库
    int gettexthigh();  //得到字符的高度
    virtual int gettextwide(char *str);  //得到一个字符串的宽度
    virtual void outtextxy(int x,int y,char *string);  //在指定位置显示字符串
    void outtext(char *string);  //在当前位置显示字符串
    void putxy(int x,int y);  //设置显示字符串的当前位置
    void getxy(int &x,int &y);  //获得当前位置
    void printf(char *fmt, ...);  //在当前位置格式化输出字符串
    void sethollow(int K) { hollow=K; }  //设置轮廓矢量字符的空心显示
    virtual int getcharwide(unsigned char c);  //得到一个字符的宽度
    virtual int outchar(int x,int y,unsigned char *cp) = 0; //在指定位置输出一个字符
protected:
    static int TEXT_X,TEXT_Y;  //字符串显示的当前位置
    int type;  //字库类型
    int org_high,org_wide;  //字符的原始高度和宽度
    int high,wide,xi;  //字符显示的高度、宽度、间隔
    FONT *fonts[3];  //三个字符集字库的字符显示类对象的指针，其中一个必为其本身
      //fonts[0]对应于ASCII字符集
      //fonts[1]对应于汉字图形字符集
      //fonts[2]对应于汉字基本字符集
    VGABASE *vga;  //图形模式类对象的指针，其具体对应的图形模式在运行时动态确定
    int hollow;  //是否作空心显示

    void *getbuf(long offset,int ll);  //从字库中取出一段数据
    void freebuf(void *buf);  //释放所取出的数据
    long getoffsetSmall(unsigned char *cp,int isize);
      //获得小字库中字形数据或指针数据的地址
    } ;

/*  ASCII字符显示基类  */
class FONT_ASC : public FONT
    {
public:
    FONT_ASC(char *name,int whe,VGABASE *v);
    int gettextwide(char *string);
    void outtextxy(int x,int y,char *string);
```

```
    } ;

/*  16点阵ASCII字库  */
class ASC16 : public FONT_ASC
    {
public:
    ASC16(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
    } ;

/*  24点阵ASCII字库  */
class ASC24 : public FONT_ASC
    {
public:
    ASC24(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
    } ;

/*  16点阵汉字库  */
class HZ16 : public FONT
    {
public:
    HZ16(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
protected:
    virtual void *getdata(unsigned char *cp);
    } ;

/*  16点阵小汉字库  */
class HZ16_S : public HZ16
    {
public:
    HZ16_S(char *name,int whe,VGABASE *v) : HZ16(name,whe,v) { }
protected:
    void *getdata(unsigned char *cp)
        { return getbuf( getoffsetSmall(cp,32), 32); }
    };

/*  24点阵基本汉字库  */
class HZ24 : public FONT
    {
public:
    HZ24(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
protected:
    virtual void *getdata(unsigned char *cp);
    } ;

/*  24点阵图形汉字库  */
```

```
class TX24 : public HZ24
    {
public:
    TX24(char *name,int whe,VGABASE *v) : HZ24(name,whe,v)
        { type=TXfont; }
protected:
    void *getdata(unsigned char *cp);
    } ;

/*  24点阵小汉字库  */
class HZ24_S : public HZ24
    {
public:
    HZ24_S(char *name,int whe,VGABASE *v) : HZ24(name,whe,v)
        { fonts[1]=this; }
protected:
    void *getdata(unsigned char *cp)
        { return getbuf( getoffsetSmall(cp,72), 72); }
    } ;

/*  32点阵基本汉字库  */
class HZ32 : public FONT
    {
public:
    HZ32(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
protected:
    virtual void *getdata(unsigned char *cp);
    } ;

/*  32点阵图形汉字库  */
class TX32 : public HZ32
    {
public:
    TX32(char *name,int whe,VGABASE *v) : HZ32(name,whe,v)
        { type=TXfont; }
protected:
    void *getdata(unsigned char *cp);
    } ;

/*  32点阵小汉字库  */
class HZ32_S : public HZ32
    {
public:
    HZ32_S(char *name,int whe,VGABASE *v) : HZ32(name,whe,v)
        { fonts[1]=this; }
protected:
    void *getdata(unsigned char *cp)
        { return getbuf( getoffsetSmall(cp,128), 128); }
```

```
    } ;

/*  40点阵基本汉字库  */
class HZ40 : public FONT
    {
public:
    HZ40(char *name,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
protected:
    virtual void *getdata(unsigned char *cp);
    } ;

/*  40点阵图形汉字库  */
class TX40 : public HZ40
    {
public:
    TX40(char *name,int whe,VGABASE *v) : HZ40(name,whe,v)
        { type=TXfont; }
protected:
    void *getdata(unsigned char *cp);
    } ;

/*  40点阵小汉字库  */
class HZ40_S : public HZ40
    {
public:
    HZ40_S(char *name,int whe,VGABASE *v) : HZ40(name,whe,v)
        { fonts[1]=this; }
protected:
    void *getdata(unsigned char *cp)
        { return getbuf( getoffsetSmall(cp,180), 180); }
    } ;

/*  BC矢量字库  */
class ASCBC : public FONT_ASC
    {
public:
    ASCBC(char *filename,int whe,VGABASE *v);
    int gettextwide(char *string);
    int getcharwide(unsigned char c);
    int outchar(int x,int y,unsigned char *cp);
protected:
    void *getdata(int &wi,unsigned char c);
    } ;

/*  ASCII轮廓矢量字库  */
class ASCSL : public FONT_ASC
    {
public:
```

```
    ASCSL(char *filename,int whe,VGABASE *v);
    int gettextwide(char *string);
    int getcharwide(unsigned char c);
    int outchar(int x,int y,unsigned char *cp);
protected:
    void *getdata(int &wi,unsigned char c);
    } ;

/*  矢量基本汉字库  */
class HZUC : public FONT
    {
public:
    HZUC(char *filename,int whe,VGABASE *v);
    int outchar(int x,int y,unsigned char *cp);
protected:
    int *getdata(unsigned char *cp);
    int *translate(int n,unsigned char *data);
    void enlarge(int x0,int y0,int *xy);
    virtual long getoffsetUC(unsigned char *cp);
    } ;

/*  小矢量基本汉字库  */
class HZUC_S :public HZUC
    {
public:
    HZUC_S(char *name,int whe,VGABASE *v) : HZUC(name,whe,v) { }
protected:
    long getoffsetUC(unsigned char *cp)
        { return getoffsetSmall(cp,6); }
    } ;

/*  矢量楷体基本汉字库  */
class HZUCK : public HZUC
    {
public:
    HZUCK(char *name,int whe,VGABASE *v) : HZUC(name,whe,v) { }
    int outchar(int x,int y,unsigned char *cp);
    } ;

/*  小矢量楷体基本汉字库  */
class HZUCK_S :public HZUCK
    {
public:
    HZUCK_S(char *name,int whe,VGABASE *v) : HZUCK(name,whe,v) { }
protected:
    long getoffsetUC(unsigned char *cp)
        { return getoffsetSmall(cp,6); }
    } ;
```

```
/*  矢量图形汉字库  */
class TXUC : public HZUCK
    {
public:
    TXUC(char *name,int whe,VGABASE *v);
protected:
    long getoffsetUC(unsigned char *cp);
    } ;

/*  小矢量图形汉字库  */
class TXUC_S :public TXUC
    {
public:
    TXUC_S(char *name,int whe,VGABASE *v) : TXUC(name,whe,v) { }
protected:
    long getoffsetUC(unsigned char *cp)
        { return getoffsetSmall(cp,6); }
    } ;
```

11.5 点阵字符显示程序

这里给出字符显示的基础程序及点阵字符显示程序。

系统程序11-2 FONT.CPP

```
#define MAXFONTHIGH 2880  //所允许的字符最大高度
#define MAXFONTWIDE 2880  //所允许的字符最大宽度

int FONT::TEXT_X=0;
int FONT::TEXT_Y=0;

/*************************************************************************
功能：构造函数。装载字库
输入参数：name＝字库文件名
          whe＝字库装载区域。其助记符同系统程序9-12
               1  装于常规内存
               2  装于扩展内存
               3  装于硬盘，即保留在硬盘上
          V＝与当前图形模式相对应的一个图形模式类指针
*************************************************************************/
FONT::FONT(char *name,int whe,VGABASE *v)
{
FILE *fp;
struct ffblK fblK;

if(findfirst(name,&fblK,0)==-1)  //字库不存在
    {
    where=inNONE;
```

```
    //设置字库未装载的标志，应用程序可通过此标志来判别所定义的字符显示类对象是否可用
    leng=0L;
    goto QUIT;
    }
leng=fblK.ff_fsize;

/*  以下22行，检测各存储区域的容量，若容量不够则转换字库的装载区域  */
if(whe==inMEM)  //如果指定将字库装于常规内存
    {
    long l;
    l=coreleft();  //获取常规内存自由空间的大小
    if(leng>=l)  //若字库大于自由的常规内存
        whe=inXMS;  //转到装于扩展内存
    if(leng>=0xffffL)  //若字库大于64K
        whe=inXMS;  //转到装于扩展内存
    }
if(whe==inXMS)  //如果字库要装于扩展内存
    {
    if(XMS::OK==0)  //如果扩展内存或其管理程序不存在
        whe=inHD;  //转到装于硬盘
    else
        {
        long l=0;
        l=(long)XMS::largestblocK()-1L;  //获取最大自由扩展内存块的大小
        l *= 1024L;
        if( leng>=l )  //若字库大于最大自由扩展内存块
        whe=inHD;  //转到装于硬盘
        }
    }
/*  以下条件语句，进行各种区域的字库装载，所装载字库的指针放到FONT::handle中  */
if(whe==inMEM)  //装于常规内存
    {
    handle.mem=malloc((unsigned)leng);
    fp=fopen(name,"rb");
    fread(handle.mem,(unsigned)leng,1,fp);
    fclose(fp);
    }
else if(whe==inXMS)  //装于扩展内存
    {
    long sagl=0x4000L;  //分段读入字库，每段的长度
    long saglc,offset,ll;
    int sagn,i;
    void *buf;

    ll=(leng+1023L)/1024L;
    handle.xms=new XMS((int)ll);  //分配扩展内存块
    buf=malloc((unsigned)sagl);
    sagn=(int)(leng/sagl);
    saglc=leng-sagl*(long)sagn;
```

```
    fp=fopen(name,"rb");
    offset=0L;
    /*  以下循环，分段读入字库，放到扩展内存  */
    for(i=0;i<sagn;i++)
        {
        fread(buf,(unsigned)sagl,1,fp);
        handle.xms->put((void *)offset,buf,sagl);
        offset+=sagl;
        }
    fread(buf,(unsigned)saglc,1,fp);  //读入字库的最后一段
    handle.xms->put((void *)offset,buf,saglc);  //放入扩展内存
    free(buf);
    fclose(fp);
    }
else  //装于硬盘
    {
    handle.file=fopen(name,"rb");
    whe=inHD;
    }
QUIT:
where=whe;
vga=v;
fonts[0]=fonts[1]=fonts[2]=0;
xi=0;
hollow=0;
if(whwhere==inNONE) OK=0;
else OK=1;
}

/*  析构函数。释放所装载的字库  */
FONT::~FONT()
{
if(where==inMEM)
    free(handle.mem);
else if(where==inXMS)
    delete handle.xms;
else if(where==inHD)
    fclose(handle.file);
}


/*************************************************************************
功能：      连接进其它两个字符集字库的字符显示类，当所要显示的字符不属于本字库时，则使
            用连接进来的某个字库。
输入参数：  fpt1＝连接进来的第1个字库的字符显示类对象的指针
            fpt2＝连接进来的第2个字库的字符显示类对象的指针
返回值：无
*************************************************************************/
```

```
void FONT::join(FONT *ftp1,FONT *ftp2)
{
if(ftp1!=0)  //指针有效
    {
    fonts[ftp1->type]=ftp1;
    ftp1->setsizeOnly(high,wide);  //使字符显示大小保持一致
    }
if(ftp2!=0)  //指针有效
    {
    fonts[ftp2->type]=ftp2;
    ftp2->setsizeOnly(high,wide);
    }
}

/*************************************************************************
功能：切断本字库与某个已连接进来的字库间的连接
输入参数：ftp＝所要切断连接字库的字符显示类对象的指针
返回值：无
*************************************************************************/
void FONT::cut(FONT *ftp)
{
int i;
for(i=0;i<3;i++)
    if(fonts[i]!=this&&fonts[i]==ftp)
        fonts[i]=0;
}

/*************************************************************************
功能：设置字符显示的大小和间隔
输入参数：  hi＝字符高度
            wi＝字符宽度
            xxi＝字符间隔
返回值：无
    所设置的字符大小仅对矢量字库有用，点阵字库只能按字库的原始大小输出字符。
*************************************************************************/
void FONT::setsizeOnly(int hi,int wi,int xxi)
{
/*  设置字符高度  */
if(hi==0)  //若所给字符高度为0
    high=org_high;  //恢复到字符的原始高度
else if(hi>0&&hi<MAXFONTHIGH)  //若所给高度在正常范围内
    high=hi;  //设置字符高度
/*  设置字符宽度  */
if(wi==0)
    wide=org_wide;
else if( wi>0&&wi<MAXFONTWIDE )
    wide=wi;
xi=xxi;  //设置两个字符间的间隔
}
```

```
/*  设置字符的大小和间隔，将同时对连接进来的字库进行设置  */
void FONT::setsize(int hi,int wi,int xxi)
{
int i;
setsizeOnly(hi,wi);
for(i=0;i<3;i++)
    if(fonts[i]!=0&&fonts[i]!=this)
        fonts[i]->setsizeOnly(high,wide);
xi=xxi;
}

/*  得到字符的高度，以象素点计  */
int FONT::gettexthigh()
{
if(OK==0) return(0);
return(high);
}

/*  得到某个字符串的宽度，以象素点计  */
int FONT::gettextwide(char *string)
{
if(OK==0) return(0);
unsigned char *str;
int wide=0;
int i=0,n;

str=(unsigned char *)string;
n=strlen(str);
for(i=0;i<n;i++)
    {
    if(str[i]<0xa1)  //如果是ASCII字符
        {
        if(fonts[0]!=0)
            wide+=fonts[0]->getcharwide(str[i]);
        }
    else if(str[i]<0xb0)  //如果是汉字图形字符
        {
        if(fonts[1]!=0)
            wide+=fonts[1]->getcharwide(str[i]);
        i++;
        }
    else  //如果是汉字基本字符
        {
        if(fonts[2]!=0)
            wide+=fonts[2]->getcharwide(str[i]);
        i++;
        }
    }
return(wide);
```

```
}

/*  获得一个字符的宽度。这里所实现的这一函数仅对等宽体字库有效  */
int FONT::getcharwide(unsigned char c)
{
if(OK==0) return(0);
unsigned char c1=c;
c=c1;
return(org_wide+xi);
}

/*  设置字符显示的当前位置  */
void FONT::putxy(int x,int y)
{
if(x>=0&&x<vga->WIDE)
    TEXT_X=x;
if(y>=0&&y<vga->HIGH)
    TEXT_Y=y;
}

/*  获得字符显示的当前位置  */
void FONT::getxy(int &x,int &y)
{
x=TEXT_X;
y=TEXT_Y;
}

/*************************************************************************
功能：在一定位置输出一个字符串
输入参数：  x0＝字符串左上角的横坐标值
            y0＝字符串左上角的纵坐标值
            string＝字符串
返回值：无
    该函数没有直接调用其本身的字符输出函数，而是通过与三个指针来调用三个字符集字库的字符输出函数，由此实现对多字符集字符串的输出。
*************************************************************************/
void FONT::outtextxy(int x0,int y0,char *string)
{
if(OK==0) return;
unsigned char *str;
int i,n;

str=(unsigned char *)string;
n=strlen(str);
for(i=0;i<n;i++)
    {
    if(str[i]<0xa1)  //如果为ASCII字符
        {
        if(fonts[0]!=0)  //
```

```
            x0+=fonts[0]->outchar(x0,y0,str+i);
        }
    else if(str[i]<0xb0)  //如果为汉字图形字符
        {
        if(fonts[1]!=0)
            x0+=fonts[1]->outchar(x0,y0,str+i);
        i++;
        }
    else  //如果为汉字标准字符
        {
        if(fonts[2]!=0)
            x0+=fonts[2]->outchar(x0,y0,str+i);
        i++;
        }
    if(x0>=vga->WIDE)
        {
        x0=0; y0+=high;
        }
    }
putxy(x0,y0);  //设置字符显示的当前位置为该字符串的右上角
}

/*  在字符显示的当前位置输出一字符串。其通过调用上一函数实现  */
void FONT::outtext(char *string)
{
outtextxy(TEXT_X,TEXT_Y,string);
}

/*************************************************************************
功能:       在当前位置格式化输出一字符串
            其输入参数的格式及含义与C语言中printf()函数完全相同。
*************************************************************************/
void FONT::printf(char *fmt, ...)
{
char *string;
va_list argptr;

string=(char *)malloc(512);
va_start(argptr, fmt);
vsprintf(string, fmt, argptr);
va_end(argptr);
outtextxy(TEXT_X,TEXT_Y,string);
free(string);
}

/*************************************************************************
功能：从字库中获取一段数据
输入参数:   offset＝数据在字库中的位置
            ll＝数据的长度
```

```
返回值：void*＝一个常规内存块的指针，该内存块存放着所读出的数据
    这是一个从存储于各种区域的字库中读取数据的统一接口，该函数使得对字库的操作与字库的
存储区域无关。
**********************************************************************/
void *FONT::getbuf(long offset,int ll)
{
void *data;
long dl=leng-offset;

if(offset>=leng)  //数据段的开始位置超出了字库的范围
    return(0);

if(dl<(long)ll)  //数据段的结束位置超出了字库的范围
    ll=(int)dl;  //减短数据
if(where==inMEM)  //如果字库在常规内存
    data=(char *)handle.mem+offset;  //直接取得数据段的指针
else
    {
    data=malloc(ll);
    if(where==inXMS)  //如果字库在扩展内存
        handle.xms->get(data,(void *)offset,ll);  //将数据读到常规内存
    else if(where==inHD)  //如果字库在硬盘
        {
        fseeK(handle.file,offset,0);
        fread(data,ll,1,handle.file);  //将数据读到常规内存
        }
    }
return(data);
}

/*  释放从字库中所读出的一段数据  */
void FONT::freebuf(void *buf)
{
if(where!=inMEM&&buf!=0)  //如果字库不在常规内存，且数据指针有效
    {
    free(buf);
    buf=0;
    }
}

/**********************************************************************
功能：获得小汉字库中某个字符的字形数据或指针数据的存放位置
输入参数：  cp＝两个字节的汉字机内码
            isize＝字形数据或指针数据的长度
返回值：long＝数据的存放位置
**********************************************************************/
long FONT::getoffsetSmall(unsigned char *cp,int isize)
{
unsigned *idx,cc;
```

```
int beg,end,i,K,n;
long offset;

idx=(unsigned *)cp;
cc=*idx;
idx=(unsigned *)getbuf(0L,2);
n=*idx;
freebuf(idx);
idx=(unsigned *)getbuf(0,2*n);
beg=1; end=n;
i=(n+1)/2;
K=0;
/*  以下循环，在小字库的索引区中查找所给定的字符，结果放在i中  */
while(1) {
    if(end-beg<=1)
        {
        K++;
        if(K>2)  //未找到该字符
            { i=0; breaK; }
        }
    if(cc==idx[i])
        breaK;
    else if(cc>idx[i])
        {
        beg=i;
        i=(beg+end+1)/2;
        }
    else
        {
        end=i;
        i=(beg+end)/2;
        }
    }
freebuf(idx);
offset=2L+(long)n*2L+(long)isize*(long)i;  //计算数据的存放位置
return(offset);
}

/*  ASCII字符显示类构造函数  */
FONT_ASC::FONT_ASC(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
fonts[0]=this;
type=ASCfont;
}

/*  获得一个ASCII字符串的宽度。仅对等宽体ASCII字库有效  */
int FONT_ASC::gettextwide(char *string)
{
if(OK==0) return(0);
```

```
int n=strlen(string);
return(n*(org_wide+xi));
}

/*  输出一个ASCII字符串。该函数可以输出与汉字机内码相重合的那部分ASCII字符。  */
void FONT_ASC::outtextxy(int x0,int y0,char *string)
{
if(OK==0) return;
unsigned char *str;
int i;

str=(unsigned char *)string;
while((*str)!=0) {
    x0+=outchar(x0,y0,str);
    str++;
    if(x0>=vga->WIDE)
        {
        x0=0;
        y0+=high;
        }
    }
putxy(x0,y0);
}

/*  16点阵ASCII字符显示类构造函数  */
ASC16::ASC16(char *name,int whe,VGABASE *v) : FONT_ASC(name,whe,v)
{
high=org_high=16;
wide=org_wide=8;
}

/*************************************************************************
功能：输出一个16×8点阵ASCII字符
输入参数：x0＝字符左上角的横坐标
          y0＝字符左上角的纵坐标
          cp＝字符的ASCII码
返回值：int＝字符的宽度加间隔
*************************************************************************/
int ASC16::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,x;
long l;

l=(long)(cp[0])*16L;
data=(unsigned char *)getbuf(l,16);
p=data;
for(i=0;i<16;i++)
```

```
    {
    c=*p;
    p++;
    x=x0;
    for(j=0;j<8;j++)
        {
        if(c&0x80)
            vga->putpixel(x,y0);  //调用某个图形模式类的写点函数实现字符显示
        x++;
        c<<=1;
        }
    y0++;
    }
freebuf(data);
return(org_wide+xi);
}

/*  24点阵ASCII字符显示类构造函数  */
ASC24::ASC24(char *name,int whe,VGABASE *v) : FONT_ASC(name,whe,v)
{
high=org_high=24;
wide=org_wide=12;
}

/*************************************************************************
功能：输出一个24×12点阵ASCII字符
输入参数：  x0＝字符左上角的横坐标
            y0＝字符左上角的纵坐标
            cp＝字符的ASCII码
返回值：int＝字符的宽度加间隔
*************************************************************************/
int ASC24::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,K,x;
long l;

l=(long)(cp[0])*48L;
data=(unsigned char *)getbuf(l,48);
p=data;
for(i=0;i<24;i++)
    {
    x=x0;
    for(j=0;j<2;j++)
        {
        c=*p;
        p++;
        for(K=0;K<8;K++)
```

```
            {
            if( c & 0x80 )
                vga->putpixel(x,y0);
            x++;
            if(x==12) breaK;
            c<<=1;
            }
        if(x==12) breaK;
        }
    y0++;
    }
freebuf(data);
return(org_wide+xi);
}

/*  16点阵汉字字符显示类构造函数  */
HZ16::HZ16(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
high=org_high=16;
wide=org_wide=16;
fonts[2]=fonts[1]=this;
  //因该字库包含了汉字基本字符集和汉字图形字符集，因此有两个连接指针指向它自身
type=HZfont;
}

/*************************************************************************
功能：输出一个16点阵汉字字符
输入参数：  x0＝字符左上角的横坐标
            y0＝字符左上角的纵坐标
            cp＝字符的ASCII码
返回值：int＝字符的宽度加间隔
*************************************************************************/
int HZ16::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,K,x;

data=(unsigned char *)getdata(cp);  //获取字形数据
/*  以下，由字形数据画出字形图象  */
p=data;
for(i=0;i<16;i++)
    {
    x=x0;
    for(j=0;j<2;j++)
        {
        c=*p;
        p++;
        for(K=0;K<8;K++)
```

```
            {
            if( c & 0x80 )
                vga->putpixel(x,y0);
            x++;
            c<<=1;
            }
        }
    y0++;
    }
freebuf(data);
return(org_wide+xi);
}

/*************************************************************************
功能：获取16点阵汉字库中某个字符的字形数据
输入参数：cp＝汉字字符两字节的机内码
返回值：void*＝所读出的字形数据块的指针
*************************************************************************/
void *HZ16::getdata(unsigned char *cp)
{
long offset;
offset=32L*(((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,32);
}

/*  24点阵汉字字符显示类构造函数  */
HZ24::HZ24(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
high=org_high=24;
wide=org_wide=24;
fonts[2]=this;
type=HZfont;
}

/*  输出一个24点阵汉字字符  */
int HZ24::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,K,y;

data=(unsigned char *)getdata(cp);  //获取字形数据
/*  以下，由字形数据画出字形图象  */
p=data;
for(i=0;i<24;i++)
    {
    y=y0;
    for(j=0;j<3;j++)
        {
```

```
        c=*p;
        p++;
        for(K=0;K<8;K++)
            {
            if( c & 0x80 )
                vga->putpixel(x0,y);
            y++;
            c<<=1;
        }
    }
    x0++;
    }
freebuf(data);
return(org_wide+xi);
}

/*  获取24点阵基本汉字库中的字形数据  */
void *HZ24::getdata(unsigned char *cp)
{
long offset;
offset=72L*(((long)cp[0]-0xb0L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,72);
}

/*  获取24点阵图形汉字库中的字形数据  */
void *TX24::getdata(unsigned char *cp)
{
long offset;
offset=72L*(((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,72);
}

/*  32点阵汉字字符显示类构造函数  */
HZ32::HZ32(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
high=org_high=32;
wide=org_wide=32;
fonts[2]=this;
type=HZfont;
}

/*  输出一个32点阵汉字字符  */
int HZ32::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,K,x,y;

data=(unsigned char *)getdata(cp);  //获取字形数据
```

```
/*  以下，由字形数据画出字形图象  */
p=data;
x=x0;
for(j=0;j<32;j++)
    {
    c=*p;
    p++;
    y=y0;
    for(K=0;K<8;K++)
        {
        if( c & 0x80 )
            vga->putpixel(x,y);
        y++;
        c<<=1;
        }
    x++;
    }
x=x0;
for(i=0;i<32;i++)
    {
    y=y0+8;
    for(j=0;j<3;j++)
        {
        c=*p;
        p++;
        for(K=0;K<8;K++)
            {
            if( c & 0x80 )
                vga->putpixel(x,y);
            y++;
            c<<=1;
            }
        }
    x++;
    }
freebuf(data);
return(org_wide+xi);
}

/*  获取32点阵基本汉字库中的字形数据  */
void *HZ32::getdata(unsigned char *cp)
{
long offset;
offset=128L*(((long)cp[0]-0xb0L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,128);
}

/*  获取32点阵图形汉字库中的字形数据  */
void *TX32::getdata(unsigned char *cp)
```

```
{
long offset;
offset=128L*(((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,128);
}

/*  40点阵汉字字符显示类构造函数  */
HZ40::HZ40(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
high=org_high=40;
wide=org_wide=36;
fonts[2]=this;
type=HZfont;
}

/*  输出一个40点阵汉字字符  */
int HZ40::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
unsigned char c,*p,*data;
int i,j,K,x,y;

data=(unsigned char *)getdata(cp);  //获取字形数据
/*  以下，由字形数据画出字形图象  */
p=data;
x=x0;
for(i=0;i<36;i++)
    {
    y=y0;
    for(j=0;j<2;j++)
        {
        c=*p;
        p++;
        for(K=0;K<8;K++)
            {
            if( c & 0x80 )
                vga->putpixel(x,y);
            y++;
            c<<=1;
            }
        }
    x++;
    }
x=x0;
for(i=0;i<36;i++)
    {
    y=y0+16;
    for(j=0;j<3;j++)
        {
```

```
        c=*p;
        p++;
        for(K=0;K<8;K++)
            {
            if( c & 0x80 )
                vga->putpixel(x,y);
            y++;
            c<<=1;
            }
        }
    x++;
    }
freebuf(data);
return(org_wide+xi);
}

/*  获取40点阵基本汉字库中的字形数据  */
void *HZ40::getdata(unsigned char *cp)
{
long offset;
offset=180L*(((long)cp[0]-0xb0L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,180);
}

/*  获取40点阵图形汉字库中的字形数据  */
void *TX40::getdata(unsigned char *cp)
{
long offset;
offset=180L*(((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L);
return getbuf(offset,180);
}
```

11.6 矢量字符显示程序

系统程序11-3 FONTSL.CPP

```
/*  BC矢量字符显示类构造函数  */
ASCBC::ASCBC(char *name,int whe,VGABASE *v) : FONT_ASC(name,whe,v)
{
unsigned char *p;
p=(unsigned char *)getbuf(0x88L,2);  //FONT类，FONT_ASC类的构造函数已先执行
org_high=high=p[0];
org_wide=wide=p[0];
freebuf(p);
}

/*  获取BC矢量字库中ASCII字符串的宽度  */
```

```
int ASCBC::gettextwide(char *string)
{
if(OK==0) return(0);
unsigned char *str,*p,bc,ec,c;
int cn,wi=0,off,n=0;
long l;

p=(unsigned char *)getbuf(0x80L,1024);
bc=p[4];
ec=p[4]+p[1]-1;
off=0x10+(int)p[1]*2-(int)p[4];
str=(unsigned char *)string;
c=*str;
while(c!=0) {
    if(c<bc||c>ec)
        wi=0;
    else
        {
        wi += p[off+c]+xi;
        n++;
        }
    str++;
    c=*str;
    }
freebuf(p);
l=wi*wide+org_wide/2;
wi=l/org_wide;
return(wi+n*xi);
}

/*  获取BC矢量字库中一个ASCII字符的宽度  */
int ASCBC::getcharwide(unsigned char c)
{
if(OK==0) return(0);
unsigned char *p,bc,ec;
int cn,wi=0,off;
long l;

p=(unsigned char *)getbuf(0x80L,1024);
bc=p[4];
ec=p[4]+p[1]-1;
if(c<bc||c>ec)
    wi=0;
else
    {
    off=0x10+(int)p[1]*2-(int)p[4];
    wi = p[off+(int)c];
    }
freebuf(p);
```

```
l=wi*wide+org_wide/2;
wi=l/org_wide;
return(wi+xi);
}

/*************************************************************************
功能：获取BC矢量字库中一个字符的字形数据及字符宽度
输入参数：  wi＝用于存放返回的宽度值
            c＝字符的ASCII码
返回值：void*＝字形数据的指针
*************************************************************************/
void *ASCBC::getdata(int &wi,unsigned char c)
{
unsigned char bc,*p;
int offset,K1,*Kp;
long l;

p=(unsigned char *)getbuf(0x80L,1024);
if( c<p[4] || c>(p[4]+p[1]-1) )  //如果字符不包含在字库中
    {
    freebuf(p);
    return(0);
    }
else
    {
    wi=p[ 0x10+(int)p[1]*2+(int)c-(int)p[4] ];
    Kp=(int *)(p+5);
    offset=*Kp+0x80;
    K1=0x10+(int)(c-p[4])*2;
    Kp=(int *)(p+K1);
    offset += (*Kp);
    freebuf(p);
    p=(unsigned char *)getbuf(offset,1024);
    return(p);
    }
}

/*  输出一个BC矢量字符  */
int ASCBC::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
int wi,yf,yd,yd2,xf,xd,xd2;
unsigned char *data,c1,c2;
int x,y,i;
long l;

data=(unsigned char *)getdata(wi,cp[0]);  //获取字形数据，放在data中
if(data==0)
    return(0);
```

```
yf=high; yd=org_high; yd2=yd/2;
xf=wide; xd=org_wide; xd2=xd/2;
y0+=high;
i=0;
while(1) {
    c1=data[i]; i++;
    c2=data[i]; i++;
    x=c1&0x7f;
    y=c2&0x7f;
    if( y>=64 ) y-=128;
    if( x>=64 ) x-=128;
    if( xf!=xd )
        { l=x*xf+xd2; x=l/xd; }  //对字符进行x方向的缩放
    if( yf!=yd )
        { l=y*yf+yd2; y=l/yd; }  //对字符进行y方向的缩放
    x+=x0;
    y=y0-y;
    if( (c1&0x80) && (c2&0x80) )
        vga->lineto(x,y);
    else if( (c1&0x80) && (c2&0x80)==0 )
        vga->moveto(x,y);
    else if( (c1&0x80)==0 && (c2&0x80)==0 )
        breaK;
    }
freebuf(data);
l=wi*xf+xd2;
wi=l/xd;
return(wi+xi);
}

/*  ASCII轮廓矢量字符显示类构造函数  */
ASCSL::ASCSL(char *name,int whe,VGABASE *v) : FONT_ASC(name,whe,v)
{
org_high=high=96;
org_wide=wide=96;
}

/*  获取一个ASCII矢量字符串的宽度  */
int ASCSL::gettextwide(char *string)
{
if(OK==0) return(0);
unsigned char *p,*str,c;
int wi=0,off,n=0;
long l;

p=(unsigned char *)getbuf(160L,128);
str=(unsigned char *)string;
c=*str;
while(c!=0) {
```

```
    if(c>=32&&c<127)
        { wi+=p[c]; n++; }
    str++;
    c=*str;
    }
freebuf(p);
l=(long)wi*wide+org_wide/2;
wi=l/org_wide;
return(wi+xi*n);
}

/*  获取一个ASCII矢量字符的宽度  */
int ASCSL::getcharwide(unsigned char c)
{
if(OK==0) return(0);
unsigned char *p;
int wi,off;
long l;
if(c<32||c>127)
          return(0);
off=192+(int)(c-32);
p=(unsigned char *)getbuf(off,2);
wi=p[0];
freebuf(p);
l=(long)wi*wide+org_wide/2;
wi=l/org_wide;
return(wi+xi);
}

/*  输出一个ASCII矢量字符串  */
int ASCSL::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
long yf,yd,yd2,xf,xd,xd2;
unsigned char *data;
int i,j,*xy,n,wi;
long l;

data=(unsigned char *)getdata(wi,cp[0]);  //获取字形数据
if(data==0)  //如果数据无效
    return(0);

xy=(int *)calloc(512,2);
yf=high; yd=org_high; yd2=yd/2;
xf=wide; xd=org_wide; xd2=xd/2;

i=0;
/*  以下循环，对字符进行缩放  */
while(1) {
```

```
    xy[i]=n=(int)data[i];
    i++;
    if(n==0)
        breaK;
    for(j=0;j<n;j++)
        {
        l=(long)data[i]*xf+xd2;
        xy[i]=l/xd; xy[i]+=x0; i++;
        l=(long)data[i]*yf+yd2;
        xy[i]=l/yd; xy[i]+=y0; i++;
        }
    }
if( hollow==0 )  //如果不作空心显示
    vga->polyfill(xy);  //填充字符内部
vga->poly(xy);  //画字符边框
freebuf(data);
free(xy);
l=(long)wi*(long)xf+xd2;
wi=l/xd;
return(wi+xi);
}

/*  获取ASCII矢量字符的字形数据及字符宽度  */
void *ASCSL::getdata(int &wi,unsigned char c)
{
unsigned char *p;
int off,i;
if(c<32||c>=127)  //如果字符不包括在字库中
    return(0);
p=(unsigned char *)getbuf(0L,288);
i=((int)c-32)*2;
off=(int)p[i]+(int)p[i+1]*256;
i=160+(int)c;
wi=p[i];
freebuf(p);
return getbuf((long)off,512);
}

/*  矢量汉字字符显示类构造函数  */
HZUC::HZUC(char *name,int whe,VGABASE *v) : FONT(name,whe,v)
{
high=org_high=96;
wide=org_wide=96;
fonts[2]=this;
type=HZfont;
}

/*  输出一个矢量汉字字符。该函数仅适合于分别填充的情况  */
int HZUC::outchar(int x0,int y0,unsigned char *cp)
```

```
{
if(OK==0) return(0);
int *xy,n,*xy1;

xy=getdata(cp);  //获取字符的字形数据。该数据已转化为多边形数据格式
if(xy==0)  //如果数据无效
    return(0);
enlarge(x0,y0,xy);  //对字符进行缩放

if( hollow==0 )  //如果不作空心显示，对多个多边形分别进行填充
    {
    xy1=xy;
    while(1) {
        n=*xy1;
        xy1++;
        if(n==0)
            breaK;
        vga->polyfill(n,xy1);
        xy1 += (n+n);
        }
    }
vga->poly(xy);
free(xy);
return(wide+xi);
}

/*************************************************************************
功能：获取一个矢量汉字的字形数据
输入参数：cp＝一个汉字的两字节的机内码
返回值：int*＝字形数据，其已转化为多边形格式的数据
*************************************************************************/
int *HZUC::getdata(unsigned char *cp)
{
long *lp,offset,doff;
int *ip,le;
unsigned char *p;
int *xy;

offset=getoffsetUC(cp);  //获取一个字符的指针数据的存放位置
p=(unsigned char *)getbuf(offset,6);  //获取指针数据
lp=(long *)p;
doff=lp[0];
ip=(int *)p;
le=ip[2];
freebuf(p);
p=(unsigned char *)getbuf(doff,le);  //获取字形数据
return translate(le,p);  //转换字形数据
}
```

```
/*  获取矢量汉字库中一个字符的指针数据的存放位置  */
long HZUC::getoffsetUC(unsigned char *cp)
{
long offset=6L*(((long)cp[0]-0xb0L)*94L+(long)cp[1]-0xa1L);
return(offset);
}

/*************************************************************************
功能：将矢量汉字的原始字形数据转化为多边形格式的数据
输入参数：n＝原始字形数据的长度
          bh＝原始字形数据
返回值：int*＝转化后的字形数据
    原始字形数据的格式参见11.2.6.3节
*************************************************************************/
int *HZUC::translate(int n,unsigned char *bh)
{
unsigned char fc,Kc;
int i,j,K,fn;
int pn,*xy;
if(bh==0)
    return(0);

xy=(int *)calloc((n*2+32),2);
i=j=0;
pn=0;
while(i<n) {
    fc=bh[i]>>4;
    if(fc>=12)
        {
        pn=j;
        j++;
        xy[j]=(bh[i]-0xC0)*2+(bh[i+1]>>7);
        i++; j++;
        xy[j]=bh[i]&0x7f;
        i++; j++;
        xy[pn]=1;
        }
    else if(fc==11)
        {
        i++;
        if( bh[i]>=0x80 )
            xy[j]=xy[j-2]-bh[i]+0x80;
        else
            xy[j]=xy[j-2]+bh[i];
        i++; j++;
        if( bh[i]>=0x80 )
            xy[j]=xy[j-2]-bh[i]+0x80;
        else
            xy[j]=xy[j-2]+bh[i];
```

```
        i++; j++;
        xy[pn]++;
        }
    else if(fc==9)
        {
        Kc=bh[i]&0x0F;
        if( Kc>=0x08 )
            xy[j+1]=xy[j-1]-Kc+0x08;
        else
            xy[j+1]=xy[j-1]+Kc;
        i++;
        if( bh[i]>=0x80 )
            xy[j]=xy[j-2]-bh[i]+0x80;
        else
            xy[j]=xy[j-2]+bh[i];
        i++; j+=2;
        xy[pn]++;
        }
    else if(fc==8)
        {
        Kc=bh[i]&0x0F;
        if( Kc>=8 )
            xy[j]=xy[j-2]-Kc+8;
        else
            xy[j]=xy[j-2]+Kc;
        i++; j++;
        if( bh[i]>=0x80 )
            xy[j]=xy[j-2]-bh[i]+0x80;
        else
            xy[j]=xy[j-2]+bh[i];
        i++; j++;
        xy[pn]++;
        }
    else if(fc<=3)
        {
        fn=bh[i];
        i++;
        for(K=0;K<fn;K++)
            {
            Kc=bh[i]>>4;
            if(Kc>=8)
                xy[j]=xy[j-2]-Kc+8;
            else
                xy[j]=xy[j-2]+Kc;
            j++;
                Kc=bh[i]&0x0F;
            if(Kc>=8)
                xy[j]=xy[j-2]-Kc+8;
            else
```

```
            xy[j]=xy[j-2]+Kc;
        i++; j++;
            xy[pn]++;
        }
    }
else if(fc==4)
    {
    fn=bh[i]&0x0F;
    i++;
    for(K=0;K<fn;K++)
        {
        xy[j]=xy[j-2]+(bh[i]>>4);
        j++;
        xy[j]=xy[j-2]+(bh[i]&0x0F);
        i++; j++;
        xy[pn]++;
        }
    }
else if(fc==5)
    {
    fn=bh[i]&0x0F;
    i++;
    for(K=0;K<fn;K++)
        {
        xy[j]=xy[j-2]-(bh[i]>>4);
        j++;
        xy[j]=xy[j-2]+(bh[i]&0x0F);
        i++; j++;
        xy[pn]++;
        }
    }
else if(fc==6)
    {
    fn=bh[i]&0x0F;
    i++;
    for(K=0;K<fn;K++)
        {
        xy[j]=xy[j-2]-(bh[i]>>4);
        j++;
        xy[j]=xy[j-2]-(bh[i]&0x0F);
        i++; j++;
        xy[pn]++;
        }
    }
else if(fc==7)
    {
    fn=bh[i]&0x0F;
    i++;
    for(K=0;K<fn;K++)
```

```
            {
            xy[j]=xy[j-2]+(bh[i]>>4);
            j++;
            xy[j]=xy[j-2]-(bh[i]&0x0F);
            i++; j++;
            xy[pn]++;
            }
        }
    else
        i++;
    }
freebuf(bh);
return(xy);
}

/*************************************************************************
功能：对字符进行缩小或放大，并将相对坐标数据转换为绝对坐标数据
输入参数：  x0＝字符左上角的横坐标值
            y0＝字符左上角的纵坐标值
            xy＝多边形格式的字形数据
返回值：无
        结果数据放在xy中
*************************************************************************/
void HZUC::enlarge(int x0,int y0,int *xy)
{
int i=0,j,n;
long yf,yd,yd2,xf,xd,xd2,l;

yf=high; yd=org_high; yd2=yd/2;
xf=wide; xd=org_wide; xd2=xd/2;
while(1) {
    n=xy[i];
    i++;
    if(n<=0)
        breaK;
    for(j=0;j<n;j++)
        {
        l=(long)xy[i]*xf+xd2;
        xy[i]=l/xd; xy[i]+=x0; i++;
        l=(long)xy[i]*yf+yd2;
        xy[i]=l/yd; xy[i]+=y0; i++;
        }
    }
}

/*************************************************************************
功能：判断两个矩形是否存在包含关系
输入数据：  xy＝第1个矩形的4个坐标值
            bxy＝第2个矩形的4个坐标值
```

```
返回值：int＝1  存在包含关系
             0  不存在包含关系
*************************************************************************/
int inclu(int *xy,int *bxy)
{
if(xy[0]<=bxy[0]&&xy[1]<=bxy[1] && xy[2]>=bxy[2]&&xy[3]>=bxy[3] )
    return(1);
if( xy[0]>=bxy[0]&&xy[1]>=bxy[1] && xy[2]<=bxy[2]&&xy[3]<=bxy[3] )
    return(1);
return(0);
}

/*  输出一个矢量汉字字符。该函数用于11.2.6.3节中所述的第3种填充情况，该函数不能完满处
理该种情况  */
int HZUCK::outchar(int x0,int y0,unsigned char *cp)
{
if(OK==0) return(0);
int i,j,*xy,n;

xy=getdata(cp);  //获得字形数据
if(xy==0)
    return(0);
enlarge(x0,y0,xy);  //缩放字形数据

if( hollow==0 )  //如不作空心显示
    {
    int **bxy,*xy1,nn=0;
    bxy=(int **)calloc(100,sizeof(void *));  //存放各多边形的矩形外框
    for(i=0;i<100;i++)
        {
        bxy[i]=(int *)calloc(4,2);
        bxy[i][0]=bxy[i][1]=10000;
        }
    xy1=xy;
    /*  以下循环，获得个多边形的矩形外框  */
    while(1) {
        n=*xy1;
        xy1++;
        if(n==0)
            breaK;
        j=0;
        for(i=0;i<n;i++)
            {
            if(xy1[j]<bxy[nn][0]) bxy[nn][0]=xy1[j];
            if(xy1[j]>bxy[nn][2]) bxy[nn][2]=xy1[j];
            j++;
            if(xy1[j]<bxy[nn][1]) bxy[nn][1]=xy1[j];
            if(xy1[j]>bxy[nn][3]) bxy[nn][3]=xy1[j];
            j++;
```

```
            }
        nn++;
        xy1 += (n+n);
        }

    xy1=xy;
    i=0;
    while(1) {
        n=*xy1;
        xy1++;
        if(n==0)
            breaK;
        for(j=i+1;j<nn;j++)
            if( inclu(bxy[i],bxy[j]) )  //检验两个多边形的矩形外框是否存在包含关系
                breaK;
        if( j==nn )  //若一个多边形的矩形外框不与其它的任何一个存在包含关系
            vga->polyfill(n,xy1);  //单独填充此多边形
        else  //若一个多边形的矩形外框与其它某个有包含关系
           {
           xy1--;
           vga->polyfill(xy1);  //同时填充此多边形之后的各多边形
           breaK;
           }
        xy1 += (n+n);
        i++;
        }
    for(i=0;i<100;i++)
        free(bxy[i]);
    free(bxy);
    }
vga->poly(xy);
free(xy);
return(wide+xi);
}

/*  图形矢量汉字字符显示类构造函数  */
TXUC::TXUC(char *name,int whe,VGABASE *v) : HZUCK(name,whe,v)
{
fonts[1]=this;
high=org_high=128;
wide=org_wide=128;
type=TXfont;
}

/*  获取图形矢量汉字库中某个字符的指针数据的存放位置  */
long TXUC::getoffsetUC(unsigned char *cp)
{
long offset=6L*(((long)cp[0]-0xa1L)*94L+(long)cp[1]-0xa1L);
return(offset);
}
```

11.7　程序使用方式

使用字符显示程序的基本步骤如下：

①定义一个图形模式类对象A。这套字符显示程序只能用本书的VGA图形显示程序来进行字符的显示，而不能用其它的图形程序如BGI来显示字符。

②如果打算将字库装载于扩展内存，运行扩展内存初始化程序XMS::init()。

③定义一个字符显示类对象D，在定义时要给定三个参数：字库文件名，字库存储区域，&A。进行这一步骤的主要问题就是要使字符显示类与字库相匹配。若需在一个字符串中输出多个字符集的字符，则需再定义其它字符集字库的对象，然后用D.join()将它们连接起来，ASCII字符集的字库只能连接到其它字库上，而不能将其它字符集的字库连接到ASCII字库上。

④运行A.init()，初始化图形模式。

⑤用对象A进行图形显示，用对象D进行字符显示。字符显示的颜色由A.setcolor()来设置。

下面给出几个示例程序。

示例程序11-1　在640×480×256色模式下显示一行16点阵的汉字。

```
void main()
{
_640_480_256 A;
XMS::init();
HZ16 D("HZK16",inXMS,&A);  //字库装载于扩展内存
A.init();
A.setcolor(7);
D.outtextxy(0,100,"16点阵汉字库");
getch();
A.close();
}
```

示例程序11-2　进行了多个字库的连接，并使用了格式化输出函数

```
void main()
{
_1024_768_16_A;
XMS::init();
HZ24 D("C:\\UCDOS\\HZKSLSTJ",inXMS,&A);  //文件名可以带路径
TX24 D1("C:\\UCDOS\\HZKSLT",inHD,&A);
ASC24 D2("ASCSL",inMEM,&A);
D.join(D1,D2);
D.setsize(48,48,2);  //统一设置各字库字符显示的大小及间隔
A.init();
A.setcolor(7);
D.putxy(0,100);
D.printf("HZKSLSTJ的长度为：%ld Bytes",D.leng);  //格式化输出函数
getch();
A.close();
}
```

示例程序11-3　使用了一个24点阵的小汉字库HZK24.SML。

```
void main()
{
_800_600_256 A;
HZ24_S D("HZK24.SML",inMEM,&A);
ASC24 D1("ASC24",inMEM,&A);
D.join(D1);
A.setcolor(4);
D.outtextxy(0,100,"1995年1月1日");
getch();
A.close();
}
```

第 12 章　图 形 打 印

很多应用程序不仅需要在屏幕上显示图形，而且还需要在打印机上打印图形。在字符打印方面，有着标准的打印接口，C语言中的fprintf(stdprn,……)函数可以完成在任何打印机上的字符打印；在图形打印方面则不存在标准的打印接口，C语言也没有提供图形打印的函数，应用程序需要根据打印机的控制指令专门编写图形打印的基础程序，才能够获得有效的图形打印功能。这一章将介绍图形打印的有关技术，并实现一套图形打印的基础支持程序。

12.1　图 象 缓 存

12.1.1　为什么需要图象缓存

打印机只能按一定的顺序，通常是从左到右、从上到下来打印一幅图象，而大多数普通打印机都没有图象缓存区，当要直接在打印机上打印图象时，就必须严格按照打印机的打印顺序来向打印机传送图象数据，如只能从左到右来打印一条水平直线，而不能从右到左，只能从上到下来打印一条垂直直线，而不能是从下到上。当要打印的图形元素比较复杂，或者需要打印多个图形元素，更或者需要动态地确定所要打印的图形元素时，这种顺序的图象传送方式，就是很难做到或者是根本不可能做到的，因此，要想实现有效的图形打印就必须在主机系统中给打印机建立一个图象缓存区，应用程序只针对图象缓存区进行图形操作，这时的图形操作就可以随机地进行，而不必按一定的顺序进行，进行完所有的图形操作后，再将缓存区中的图象数据按一定的顺序输出到打印机，从而完成图形打印。

有一种较常用的图形打印方法称为屏幕硬拷贝，这种方法实际就是将显示存储器用作打印缓存区，当需要打印一幅图象时，就首先在屏幕上画出这幅图象，然后从显示存储器中读出图象数据，输出到打印机。这种图形打印方法有两个明显的缺点，一是，由于屏幕分辨率要低于打印机的分辨率，这种打印方法打印出的图形或者偏小或者比较粗糙，难以获得满意的效果；二是，图形打印和图形显示被联系在一起，无法进行独立而灵活的图形打印。因此，要想打印出高质量的图形，获得灵活、独立的打印功能，就必须建立专门的图形打印缓存区。

大部分激光打印机本身都带有图象缓存器并具有图形处理功能，可以直接在这类打印机上进行图形输出，而不必在主机系统中为它们建立图象缓存区，但这时就需要根据打印机所采用的图形控制语言来进行图形输出，在这类打印机上打印图形的主要问题就是要掌握其图形控制语言。本书不介绍这类打印机的图形打印技术。

12.1.2　图象缓存的实现方案

屏幕硬拷贝也有一个明显的优点，就是能直接利用图形显示中的各种绘图功能、字符显示功能，而不必再为图形打印专门提供这些功能，当要建立独立的打印缓存区时，就需要专门为图形打印提供绘图及字符显示功能，这是建立独立的打印缓存区所要解决的一个主要问题。

程序在屏幕上显示图形，实际上就是向显示存储器写入图象数据，而建立了打印缓存

区之后，程序在打印机上打印图形，也就是向打印缓存区写入图象数据，从操作方式上看，显示存储器和打印缓存区之间并不存在本质的差别，而且图形显示和图形打印所需要的功能也是基本相同的，可以把对打印缓存区的图形操作看作是一个特殊图形显示模式下的图形操作，因而在编程中，可以直接从基本图形类VAGBASE中派生出一个图形模式类，在该类中实现对打印缓存区的基本图形操作，从而直接获得VGABASE类中的绘图功能及上一章所实现的字符显示功能，而不需要专门为图形打印编写这些功能，系统程序12-1给出了该类的说明。

程序只针对黑白打印机建立图象缓存，这时图象缓存区中存储的是一幅单色图象，以象素点计的图象宽度和高度在建立缓存区时确定，它们可以是任意值，图象的宽度和高度不同，缓存区的大小也就不同。缓存区的结构为：缓存区中的每个数据位对应于图象中的一个象素点，当某位为1时，其所对应的象素点就应打印为黑色，否则就不打印该象素点。一个字节对应于水平方向连续的8个象素点，若干个字节对应于图象中的一行象素点，两条扫描线中的象素点不会包含在同一个字节中，每行象素点所占的字节数总为偶数，当图象宽度不为16的整数倍时，每条扫描线所占字节的最后若干位就没有使用。

为一个300dpi(点/英寸)的打印机建立一个A4幅面的图象缓存区，所需的空间约为960K，显然不能在常规内存中建立此缓存区，因此程序中实现了在常规内存、扩展内存及硬盘中建立打印缓存区的功能，并实现了一个统一的读、写操作接口来对这三种存储区域进行读写，从而使得能用同一组图形操作函数来操作这三种区域中的图象缓存区。

系统程序12-1 VGAPRT.H

```
/*  打印缓存图形模式类  */
class prt_buf : public VGABASE
    {
public:
    prt_buf(int wi,int hi,int whe=inMEM);  //构造函数，确定分辨率及存储区域
    ~prt_buf();  //析构函数，释放图象缓存区
    int init(void) { return(OK); }  //初始化在构造函数中完成
    void close(void) { }  //不需关闭
    /*  以下为10个与图形显示相同的图形操作函数  */
    void setcolor(unchar col);
    COLOR setcolorto(unchar col);
    void putpixel(int x,int y);
    COLOR getpixel(int x,int y);
    void scanline(int x1,int x2,int y);
    void cls0();
    void cls();
    void getscanline(int x1,int y1,int n,void *buf);
    void putscanline(int x1,int y1,int n,void *buf);
    int scanlinesize(int x1,int x2);

    unchar *getprtdata(int y);  //从图象缓存区中读取数据，供图象打印用
    unchar *buffer;  //缓存区中当前一行象素点的图象数据，处于常规内存中
    //各种存储区域的统一操作接口
    void getbuffer(int y);  //更新当前象素点行
```

```
protected:
    int where,widebyte;  //存储区域；一行象素所对应的字节数
    int bufy;  //当前象素点行
    union { unsigned char **mem;
            class XMS *xms;
            FILE *file;
          } handle;  //各种区域中的图象缓存区的通用指针
    } ;
```

12.1.3　图象缓存操作程序

系统程序12-2　PRT.CPP

```
/*************************************************************************
功能：构造函数。在一定的存储区域建立具有一定大小的图形缓存区
输入参数：  wi＝缓存区的水平象素数
            hi＝缓存区的垂直象素数
            whe＝所指定的建立缓存区的区域，该参数可缺省，若缺省则默认为常规内存
*************************************************************************/
prt_buf::prt_buf(int wi,int hi,int whe)
{
long size,l;
int i;

WIDE=wi;
HIGH=hi;
widebyte=(wi+7)/8;  //每行字节数
if( widebyte&0x01 )
    widebyte++;  //使其为偶数，加快扩展内存的操作
size=(long)widebyte*(long)hi;  //缓存区的大小

if(whe==inMEM)  //若指定建于常规内存
    {
    l=coreleft();  //自由常规内存的大小
    if(size>=(l-0x10000L))  //建立存储区后，若自由常规内存小于64K
        whe=inXMS;  //转到扩展内存
    }
if(whe==inXMS)  //若需建于扩展内存
    {
    if(XMS::OK==0)  //扩展内存或其管理程序不存在
        whe=inHD;  //转到硬盘
    else
        {
        l=(long)XMS::largestblocK()-1L;
        l *= 1024L;
        if( size>=l )  //扩展内存不够
            whe=inHD;  //转到硬盘
        }
```

```
    }
where=whe;

if(where==inMEM)  //建于常规内存
    {
    unsigned char **pp;
    pp=(unsigned char **)calloc(hi,sizeof(void *));
    for(i=0;i<hi;i++)
        pp[i]=(unchar *)calloc(widebyte,1);  //以行为单位分配内存
    handle.mem=pp;
    buffer=pp[0];
    }
else if(where==inXMS)  //建于扩展内存
    {
    l=(size+1023L)/1024L;
    handle.xms=new XMS((int)l);
    buffer=(unchar *)calloc(widebyte,1);
    l=0;
    for(i=0;i<hi;i++)
        {
        handle.xms->put((void *)l,buffer,widebyte);
        l += (long)widebyte;
        }
    }
else if(where==inHD)  //建于硬盘
    {
    FILE *fp;
    fp=fopen("prt_buf.img","w+b");  //建立一文件
    buffer=(unchar *)calloc(widebyte,1);
    for(i=0;i<hi;i++)
        fwrite(buffer,1,widebyte,fp);
    handle.file=fp;  //文件指针指向缓存区
    }
else
    buffer=(unchar *)calloc(widebyte,1);
bufy=0;
setcolor(1);
VESAmodeNo=-1;  //打印缓存区的标志
if(where==inNONE) OK=0;
else OK=1;
}

/*  析构函数。释放缓存区  */
prt_buf::~prt_buf()
{
int i;
if( where==inMEM )
    {
    for(i=0;i<HIGH;i++)
```

```
        free(handle.mem[i]);
    free(handle.mem);
    }
else
    free(buffer);
if( where==inXMS )
    delete handle.xms;
else if( where==inHD )
    fclose(handle.file);  //关闭文件，但没有删除
}

/*  设置当前颜色0或1  */
void prt_buf::setcolor(unchar col)
{
CUR_COLOR.dword=0L;
if(col>0)
    CUR_COLOR.byte=1;
}

/*  颜色转换  */
COLOR prt_buf::setcolorto(unchar col)
{
COLOR color={0};
if(col>0)
    color.byte=1;
return(color);
}

/*  在缓存区中写一个象素点  */
void prt_buf::putpixel(int x,int y)
{
if(x<0||x>=WIDE||y<0||y>=HIGH)  //进行边界检查，这是必需的，超出边界会死机
    return;

int byte,bit;
unsigned char c=0x80;
byte=x>>3;
bit=x&7;
c >>= bit;
getbuffer(y);  //将象素点所在行的数据取到常规内存，放于prt_buf::buffer中
if( CUR_COLOR.byte )
    buffer[byte] |= c;
else
    buffer[byte] &= (~c);
}

/*  读一个象素点  */
COLOR prt_buf::getpixel(int x,int y)
{
```

```
COLOR color={0L};
if(x<0||x>=WIDE||y<0||y>=HIGH)
    return(color);
int byte,bit;
unsigned char c;
byte=x>>3;
bit=x&7;
c >>= bit;
getbuffer(y);
c=buffer[byte]<<bit;
color.byte=(c>>7);
return(color);
}

/*  画扫描线  */
void prt_buf::scanline(int x1,int x2,int y)
{
if(y<0||y>=HIGH||x1>=WIDE||x2<0)
    return;
if(x1<0) x1=0;
if(x2>=WIDE) x2=WIDE-1;
int byte1,byte2,bit1,bit2,i;
unsigned char c1=0xff,c2=0xff;
if(x1>x2)
    { i=x1; x1=x2; x2=i; }
getbuffer(y);
byte1=x1>>3;
byte2=x2>>3;
bit1=x1&7;
bit2=7-(x2&7);
if( byte2>byte1 )  //始点、末点不在一个字节中
    {
    /*  画首字节中的点  */
    c1 >>= bit1;
    if( CUR_COLOR.byte )
        {
        buffer[byte1] |= c1;
        c1=0xff;
        }
    else
        {
        buffer[byte1] &= (~c1);
        c1=0;
        }
    /*  画中间字节中的点  */
    for(i=byte1+1;i<byte2;i++)
        buffer[i]=c1;
    /*  画尾字节中的点  */
    c2 <<= bit2;
```

```
    if( CUR_COLOR.byte )
        buffer[byte2] |= c2;
    else
        buffer[byte2] &= (~c2);
    }
else  //始点、末点在一个字节中
    {
    c1 >>= bit1;
    c2 <<= bit2;
    c1 = c1&c2;
    if( CUR_COLOR.byte )
        buffer[byte1] |= c1;
    else
        buffer[byte1] &= (~c1);
    }
}

/*  置零清屏  */
void prt_buf::cls0()
{
unsigned char *buf;
int i;
buf=(unsigned char *)calloc(widebyte,1);
for(i=0;i<HIGH;i++)
    {
    getbuffer(i);
    memcpy(buffer,buf,widebyte);
    }
free(buf);
}

/*  置色清屏  */
void prt_buf::cls()
{
unsigned char *buf;
int i;
buf=(unsigned char *)calloc(widebyte,1);
if( CUR_COLOR.byte==1 )
    for(i=0;i<widebyte;i++)
        buf[i]=0xff;
for(i=0;i<HIGH;i++)
    {
    getbuffer(i);
    memcpy(buffer,buf,widebyte);
    }
}

/*  读扫描线  */
void prt_buf::getscanline(int x1,int y1,int n,void *buf)
```

```
{
if(y1<0||y1>=HIGH||x1>=WIDE)
    return;
int byte1,byte2,tol,tor;
int x2,i,p=0;
unsigned char c,*cp;

x2=x1+n-1;
if(x1<0) x1=0;
if(x2>=WIDE) x2=WIDE-1;
getbuffer(y1);
cp=(unsigned char *)buf;
byte1=x1>>3;
byte2=x2>>3;
tol=x1&7;
tor=8-tol;
for(i=byte1;i<=byte2;i++)
    {
    c=buffer[i];
    cp[p] |= (c>>tor);
    p++;
    cp[p]=(c<<tol);
    }
}

/*  写扫描线  */
void prt_buf::putscanline(int x1,int y1,int n,void *buf)
{
int byte1,byte2,tol,tor,bit1,bit2,i,p=1,x2;
unsigned char c1,c2,*cp;

getbuffer(y1);
cp=(unsigned char *)buf;
x2=x1+n-1;
byte1=x1>>3;
byte2=x2>>3;
if( byte2>byte1 )  //始点、末点不在一个字节中
    {
    tor=x1&7;
    tol=8-tor;
    /*  写首字节中的点  */
    c1 = (cp[p]>>tor);
    c2 = 0xff;
    c2 <<= tol;
    buffer[byte1] &= c2;
    buffer[byte1] |= c1;
    /*  写中间字节中的点  */
    for(i=byte1+1;i<byte2;i++)
        {
```

```
        c1=(cp[p]<<tol);
        buffer[i]=c1;
        p++;
        c1=(cp[p]>>tor);
        buffer[i]|=c1;
        }
    /*  写尾字节中的点  */
    c1=(cp[p]<<tol);
    p++;
    c2=(cp[p]>>tor);
    c1=c1|c2;
    tor=x2&7;
    tol=7-tor;
    c2=0xff;
    c2 <<= tol;
    c1=c1&c2;
    c2=0xff;
    c2 >>=tor;
    buffer[byte2] &= c2;
    buffer[byte2] |= c1;
    }
else  //始点、末点在一个字节中
    {
    c1=c2=0xff;
    tol=7-(x1&7);
    c1 <<= tol;
    tor=x2&7;
    c2 >>= tor;
    c1=c1|c2;
    buffer[byte1] &= c1;
    c1=~c1;
    tor=x1&7;
    c2=(cp[p]>>tor);
    c1=c1&c2;
    buffer[byte1] |= c1;
    }
}

/*  取得保存一条扫描线所需缓冲区的大小  */
int prt_buf::scanlinesize(int x1,int x2)
{
int K=(x2-x1)/8 + 4;
if( K&1 )
    K++;
return(K);
}


/*************************************************************************
```

```
功能：取出缓存区中某一行的数据
输出参数：y＝所要取的行
返回值：无
        所取出的数据放在prt_buf::buffer中
**************************************************************************/
void prt_buf::getbuffer(int y)
{
if(y==bufy)
    return ;
long off;

if( where==inMEM )  //若缓存区在常规内存
    buffer=handle.mem[y];  //直接获得其指针
else if( where==inXMS )  //若在扩展内存
    {
    off=(long)bufy*(long)widebyte;
    handle.xms->put((void *)off,buffer,widebyte);  //首先写入当前行的数据
    off=(long)y*(long)widebyte;
    handle.xms->get(buffer,(void *)off,widebyte);  //然后读出新行的数据
    }
else if( where==inHD )
    {
    off=(long)bufy*(long)widebyte;
    fseeK(handle.file,off,0);
    fwrite(buffer,1,widebyte,handle.file);  //首先写入
    off=(long)y*(long)widebyte;
    fseeK(handle.file,off,0);
    fread(buffer,1,widebyte,handle.file);  //然后读出
    }
bufy=y;
}
```

12.2 EPSON、HP打印机上的图象打印

图象打印的任务就是将图象缓存区中的图象数据按一定的顺序输出到打印机上。图象打印的实现方式与打印机的类型密切相关，这里针对EPSON系列和HP系列的针式打印机及喷墨打印机介绍图象打印的技术和程序。

12.2.1 EPSON系列打印机

1. 概况

EPSON系列的针式打印机一直是我国普通打印机市场的主流产品，最近两年EPSON开始生产喷墨打印机，目前在我国也已有了很大的销量。EPSON打印机的型号有十几种，表12-1列出了几种主要产品的情况

表12-1 几种EPSON打印机的情况

型号	类型	分辨率/dpi	幅宽	控制指令
LQ 1600	针式	180	宽行打印纸	ESC/P
LQ 1600II	针式	360	宽行打印纸	ESC/P2
Stylus 800	喷墨	360	A4复印纸	ESC/P2
Stylus 1000	喷墨	360	A3复印纸	ESC/P2

较早期的EPSON针式打印机采用的都是ESC/P打印控制指令，较新的打印机则采用的是ESC/P2，ESC/P2完全兼容ESC/P，下面主要针对ESC/P2来介绍EPSON的图形打印技术。

2. ESC/P2图象打印指令

一条打印指令由若干个字节所组成，一条指令的第1个字节的值通常为27，它是键盘上Esc键的ASCII码。在C语言中用biosprint()函数或fprintf(stdprn,……)函数，将一条控制指令的各个字节顺序输出到打印机上，即可实现对打印机的控制。

下面给出ESC/P2指令集中与图象打印有关的控制指令，其它的指令可参见EPSON打印机的随机手册。每条指令都同时给出了其各个字节所对应的ASCII字符，各个字节的十进制值及16进制值。

(1)初始化打印机(ESC @)

ASCII字符　ESC　@
十进制值　27　64
16进制值　1B　40
使打印机状态复原，并清除前面的打印机设置指令。

(2)回车(CR)

ASCII字符　CR
十进制值　13
16进制值　0D
打印头退回到最左边。

(3)换行(LF)

ASCII字符　LF
十进制值　10
16进制值　0A
纸按当前设定的行距走一行。

(4)换页(FF)

ASCII字符　FF
十进制值　12
16进制值　0C
根据当前页长度自动走纸到下页的页顶。

(5)设定n/180英寸换行量(ESC 3)

ASCII字符　ESC　3　n
十进制值　27　51　n
16进制值　1B　33　n

设置n/180英寸换行行距，n可为0～255之间的任意值。

EPSON打印机总是以打印头的高度为单位一行行地打出图象，其打印头的高度为1/7.5英寸，而其开机后默认的行距为1/6英寸，因此进行图象打印前必须用此指令将行距设为1/7.5英寸，即24/180英寸，否则图象就会出现间断。还有其它几个行距设置指令，但通常只需使用这一个。

(6)设定单/双向打印(ESC U)

ASCII字符 ESC U n

十进制值 27 85 n

16进制值 1B 55 n

n＝1或‘1’ 选择单向打印

n＝0或‘0’ 选择双向打印

双向打印时，两个打印头行之间往往不能准确对齐，因此打印图象之前通常都应设置为单向打印，但这时打印速度会较慢。

(7)设定图象模式并输出图象数据(ESC *)

ASCII字符 ESC * m nL nH data

十进制值 27 42 m nL nH data

16进制值 1B 2A m nL nH data

m是图象模式的代码，其具体取值所对应的图象模式见表12-2。

nL，nH共同表示了所要打印图象的列数，设列数为n，则有：n=nL+256*nH。

data为若干个字节的图象数据，该数据的长度取决于列数和当前模式下的每列点数，数据长度的计算方式为：图象数据长度(字节)＝列数×每列点数/8。所输出的图象数据的长度必须严格满足要求。

该指令按一定的图象模式打印出一个打印头行的图象，打印完后不换行也不回车。一个打印头行包含有若干象素点行，具体包含的行数与图象模式有关，表12-2中的每列点数即为一个打印头行所包含的象素点行的数量。

图象数据中的每一位对应于图象中的一个象素点。每个字节对应于一列象素中连续的8个象素点，其中位7对应于8个中最上面一个，位0对应于最下面一个。当图象模式的每列点数为8时，图象中的一列象素对应于1个字节；当每列点数为24时，一列象素对应于连续的3个字节，各字节按由上到下的次序与象素点对应；当每列点数为48时，一列象素对应于连续的6个字节，各字节按由上到下的次序与象素点对应。以1个、3个或6个字节为一组从左到右依次对应于图象中的各列象素。

表12-2 EPSON打印机的图象模式

m	水平密度/dpi	垂直密度/dpi	每列点数
0	60	60	8
1	120	60	8
2	120	60	8
3	240	60	8

续表

m	水平密度/dpi	垂直密度/dpi	每列点数
4	80	60	8
6	90	60	8
32	60	180	24
33	120	180	24
38	90	180	24
39	180	180	24
40	360	180	24
71	180	360	48
72	360	360	48
73	360	360	48

其中，ESC/P指令集或较早期的打印机只能支持0～40的图象模式，而不能支持垂直密度为360dpi的那些图象模式。

3. 图象打印的步骤

下面给出用ESC/P2指令集进行图象打印的通常步骤。

①执行“ESC @”指令，进行打印机初始化；

②执行“ESC U”指令，设置单向打印；

③执行“ESC 3”指令，将行距设置为24/180英寸；

④用“ESC *”指令，以相同的图象模式依次输出各打印头行的图象数据，每打印一行图象即执行一次回车和一次换行指令；

⑤执行“ESC 3”指令，将行距还原为30/180英寸。

12.2.2　HP系列打印机

HP系列只有喷墨打印机和激光打印机，没有针式打印机。HP的喷墨打印机一直是我国喷墨打印机市场上的主流产品，其黑白喷墨打印机的主要型号是HP500Q，该打印机的分辨率为300dpi，打印幅宽为A4复印纸，其采用的打印控制指令为PCL(Printer Control Language)，也可支持ESC/P打印指令，但此时图形打印的效果不太好。HP系列的所有打印机采用的都是PCL控制指令，下面介绍使用PCL进行图象打印的方法。

1. PCL图象打印指令

(1)回车(CR)

ASCII字符　CR

十进制值　13

16进制值　0D

打印头退回到最左边。

(2)换行(LF)

ASCII字符　LF

十进制值　10
16进制值　0A
纸按当前设定的行距走一行。

(3)送出当前页

ASCII字符　ESC　&　l　0　H
十进制值　27　38　108　48　72
16进制值　1B　26　6C　30　48

(4)设置单/双向打印

ASCII字符　ESC　&　d　n　D
十进制值　27　38　107　n　87
16进制值　1B　26　6B　n　57
n＝48或‘0’ 从左至右单向打印
n＝49或‘1’ 双向打印
n＝50或‘2’ 从右至左单向打印

(5)设置图形质量

ASCII字符　ESC *　r　n　Q
十进制值　27　42　114　n　81
16进制值　1B　2A　72　n　51
n＝49或‘1’ 草稿质量
n＝50或‘2’ 准印刷质量

(6)设置图象打印的分辨率

PCL支持75dpi、100dpi、150dpi、300dpi、600dpi的图形打印，但只有一些较新的机型才能提供600dpi的分辨率，大多数打印机只能支持到300dpi，这5种分辨率的设置指令列于表12-3。每种分辨率同时代表了水平和垂直分辨率。

表12-3　PCL图形打印分辨率设置指令

分辨率	ASCII字符	十进制值	16进制值
75dpi	ESC * t 7 5 R	27 42 116 55 53 82	1B 2A 74 37 35 52
100dpi	ESC * t 1 0 0 R	27 42 116 49 48 48 82	1B 2A 74 31 30 30 52
150dpi	ESC * t 1 5 0 R	27 42 116 49 53 48 82	1B 2A 74 31 35 30 52
300dpi	ESC * t 3 0 0 R	27 42 116 51 48 48 82	1B 2A 74 33 30 30 52
600dpi	ESC * t 6 0 0 R	27 42 116 54 48 48 82	1B 2A 74 36 30 30 52

(7)传送图形

ASCII字符　ESC　*　b　nb　W　data
十进制值　27　42　98　nb　89　data
16进制值　1B　2A　62　nb　57　data

data为若干个字节的图象数据，其对应于一行象素，其字节数由nb给出。data中的各个数据位按从左到右的顺序依次对应于一行图象中的各个象素点。

nb包含多个字节，每个字节对应于一个ASCII数字字符，多个数字字符一起表示了图象数据data的长度，如指令“ESC * b 1 0 0 W data”表示所传送的图象数据的长度为100个字节。

该指令打印出一行象素点，打印完后自动转到下一象素行的最左边，准备打印下一象素行。

PCL是按象素点行逐行打印图象，而不是像ESC/P那样按打印头行一次打多行象素，PCL的这种打印方式称为光栅图象打印，使用这种方式比ESC/P要简单得多。

(8)结束图象

ASCII字符	ESC	*	r	b	C
十进制值	27	42	114	98	67
16进制值	1B	2A	72	62	43

结束图象输出。

2. 图象打印步骤

下面给出用PCL指令集进行图象打印的通常步骤。

①设置单项打印；

②设置图象打印的分辨率

③设置准印刷图形质量

④用图象传送指令，从上到下依次输出各象素行的图象数据；

⑤执行结束图象指令。

12.2.3　图象打印编程方案

程序将针对ESC/P、ESC/P2及PCL打印控制指令实现图象打印功能，对PCL，将实现其所有图象分辨率下的图象打印，对ESC/P及ESC/P2，则只实现三种水平密度和垂直密度相同的图象模式下的图象打印，这三种图象模式为60×60、180×180、360×360。

不同的打印指令集或不同的分辨率下的图象打印方式都是存在差别的，但又有很多相同之处，因此这组程序也很适合采用C++来编写。在程序中，对每个指令集下的每种分辨率定义一个打印模式类，这些类中相同的部分放在一些基类中来实现，由此形成一个如图12-1所示的图象打印类体系，该类体系中各个类的说明见系统程序12-3。

通过向一个打印模式类对象传送一个图形模式类对象的指针，来实现图象缓存区与图象打印功能之间的连接，通常传送的应是与图象存储区相对应的图形模式类prt_buf对象的指针，但也可将一个正常的图形模式类对象的指针传送给打印模式类，这时就将从显示存储器中读取所要打印的图象数据，从而实现对屏幕硬拷贝的支持，在进行屏幕硬拷贝时，屏幕上颜色值为0的象素点将打印为白色(即不打印)，颜色不为0的象素点都将打印为黑色。

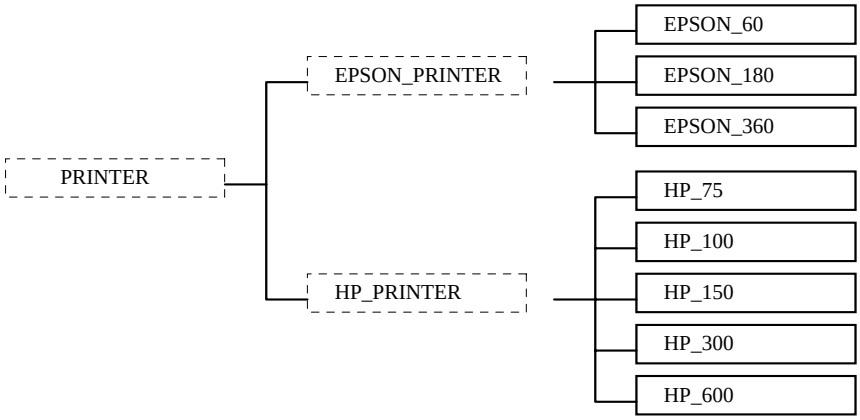

图12-1 图象打印类体系

系统程序12-3 VGAPRT.H

```
/*  图象打印基类  */
class PRINTER {
public:
    PRINTER(VGABASE *v) { vga=v; }  //构造函数，接收一个图形模式类对象的指针
    virtual void print(void)=0;  //图象打印
    virtual void outpaper(void)=0;  //换页或出纸
    void nextrow(void);  //换行
protected:
    VGABASE *vga;  //图象模式类对象的指针，该指针对应于一个图象缓存区
    int byten;  //一行图象数据的字节数
    unchar *getprtdata(int y);  //从图象缓存区中获得图象数据
    void outcode(unchar *code);  //输出一个打印控制指令
    virtual void setdpi() { };  //设置分辨率，HP打印机用
    virtual void setmode(int pxn) { }  //设置图象模式，EPSON打印机用
    } ;

/*  HP图象打印基类  */
class HP_PRINTER : public PRINTER
    {
public:
    HP_PRINTER(VGABASE *v) : PRINTER(v) { }
    void print(void);
    void outpaper(void);
protected:
    void setbytenum(int n);  //图象输出
    void setgz();  //设置准印刷图形质量
    void setdx();  //设置单向打印
    void end();  //结束图象打印
    } ;
```

```
/*  HP 600dpi打印模式类  */
class HP_600 : public HP_PRINTER
    {
public:
    HP_600(VGABASE *v) : HP_PRINTER(v) { }
protected:
    void setdpi();
    } ;

/*  HP 300dpi打印模式类  */
class HP_300 : public HP_PRINTER
    {
public:
    HP_300(VGABASE *v) : HP_PRINTER(v) { }
protected:
    void setdpi();
    } ;

/*  HP 150dpi打印模式类  */
class HP_150 : public HP_PRINTER
    {
public:
    HP_150(VGABASE *v) : HP_PRINTER(v) { }
protected:
    void setdpi();
    } ;

/*  HP 100dpi打印模式类  */
class HP_100 : public HP_PRINTER
    {
public:
    HP_100(VGABASE *v) : HP_PRINTER(v) { }
protected:
    void setdpi();
    } ;

/*  HP 75dpi打印模式类  */
class HP_75 : public HP_PRINTER
    {
public:
    HP_75(VGABASE *v) : HP_PRINTER(v) { }
protected:
    void setdpi();
    } ;

/*  EPSON图象打印基类  */
class EPSON_PRINTER : public PRINTER
    {
public:
```

```
    EPSON_PRINTER(VGABASE *v) : PRINTER(v) { }
    void print(void);
    void outpaper(void);
protected:
    int dpi;  //打印模式的分辨率
    void init(void);  //初始化打印机
    void setjg180(unchar c180=24);  //设置n/180英寸行距
    void setdx(void);  //设置单向打印
    void setmodeb(int pxn,unchar *code);  //设置图象模式的基础函数
    } ;

/*  EPSON 360dpi打印模式类  */
class EPSON_360 : public EPSON_PRINTER
    {
public:
    EPSON_360(VGABASE *v) : EPSON_PRINTER(v) { dpi=360; }
protected:
    void setmode(int pxn);
    } ;

/*  EPSON 180dpi打印模式类  */
class EPSON_180 : public EPSON_PRINTER
    {
public:
    EPSON_180(VGABASE *v) : EPSON_PRINTER(v) { dpi=180; }
protected:
    void setmode(int pxn);
    } ;

/*  EPSON 60dpi打印模式类  */
class EPSON_60 : public EPSON_PRINTER
    {
public:
    EPSON_60(VGABASE *v) : EPSON_PRINTER(v) { dpi=60; }
protected:
    void setmode(int pxn);
    } ;
```

12.2.4 图象打印程序

系统程序12-4 PRT.CPP

```
/*************************************************************************
功能：从图象缓存区中读取一行象素的图象数据
输入参数：y＝象素行
返回值：unchar *＝所读出数据的指针
    该函数即可从专门的打印缓存区中读取数据，也可从某种图形显示模式下的显示存储器中读取
数据。
*************************************************************************/
```

```
unchar *PRINTER::getprtdata(int y)
{
if(y<0||y>=vga->HIGH)  //所给行超出了图象的范围
    return(0);

if( vga->VESAmodeNo==-1 )  //如果图形模式类为prt_buf，则从专门的打印缓存区中读取数据
    {
    prt_buf *p;
    p=(prt_buf *)vga;
    p->getbuffer(y);
    return(p->buffer);
    }
else  //如果为正常的图形显示模式，从显示存储器中读取数据，即进行屏幕硬拷贝
    {
    static unsigned char *buf=0;
    unsigned char c;
    int bn,K,wi,x;

    wi=vga->WIDE;
    if( buf!=0 )
        free(buf);
    buf=(unsigned char *)calloc(byten,1);
    bn=K=0;
    c=0x80;
    for(x=0;x<wi;x++)
        {
        if( vga->getpixel(x,y).dword )  //用图形模式类的读点函数读取图象数据
            buf[bn] |= c;
        K++;
        c >>= 1;
        if( K>=8 )
            {
            K=0;
            c=0x80;
            bn++;
            }
        }
    return(buf);
    }
}

/*************************************************************************
功能：通用打印指令输出函数
输入参数：code＝打印指令的多个字节，以0结尾。
返回值：无
    如一个打印指令中包含有0，则不能使用此函数。
*************************************************************************/
void PRINTER::outcode(unsigned char *code)
{
int i=0;
while( code[i]!=0 ) {
    biosprint(0,code[i],0);
```

```
    i++;
    }
}

/*  换行并回车  */
void PRINTER::nextrow()
{
biosprint(0,0x0a,0);
biosprint(0,0x0d,0);
}

/*  HP打印机的图象打印  */
void HP_PRINTER::print()
{
int i,j;
int yn=vga->HIGH;  //所要打印图象的行数
unsigned char *buf;

byten=vga->WIDE/8;  //一行象素所用的字节数。getprtdata()函数需根据此数读取图象数据
setdx();  //设置单向打印
setdpi();  //设置分辨率
setgz();  //设置准印刷质量
/*  以下循环，逐行打印图象  */
for(i=0;i<yn;i++)
    {
    buf=getprtdata(i);  //从缓存区中获取一行象素数据
    setbytenum(byten);  //图象输出指令头
    for(j=0;j<byten;j++)  //输出图象数据
        biosprint(0,buf[j],0);
    }
end();  //结束图象打印
}

/*  出纸 */
void HP_PRINTER::outpaper()
{
outcode("\x1b\x26\x6c\x30\x48\0");
}

/*  设置单向打印  */
void HP_PRINTER::setdx()
{
outcode("\x1b\x26\x6b\x30\x57");
}

/*  设置准印刷图形质量  */
void HP_PRINTER::setgz()
{
outcode("\x1b\x2a\x72\x32\x51");
}
```

```
/*************************************************************************
功能：输出PCL图象传送指令的头部(不含图象数据)
输入参数：n＝图象数据的字节数
返回值：无
*************************************************************************/
void HP_PRINTER::setbytenum(int n)
{
int i,K;
unsigned char *str;
str=(unsigned char *)calloc(10,1);
itoa(n,str,10);
biosprint(0,0x1b,0);
biosprint(0,0x2a,0);
biosprint(0,0x62,0);
i=0;
while(str[i]!=0) {
    biosprint(0,str[i],0);
    i++;
    }
biosprint(0,0x57,0);
free(str);
}

/*  结束图象打印  */
void HP_PRINTER::end()
{
outcode("\x1b\x2a\x72\x62\x43");
}

/*  设置600dpi的图象分辨率  */
void HP_600::setdpi()
{
outcode("\x1b\x2a\x74\x36\x30\x30\x52");
}

/*  设置300dpi的图象分辨率  */
void HP_300::setdpi()
{
outcode("\x1b\x2a\x74\x33\x30\x30\x52");
}

/*  设置150dpi的图象分辨率  */
void HP_150::setdpi()
{
outcode("\x1b\x2a\x74\x31\x35\x30\x52");
}

/*  设置100dpi的图象分辨率  */
void HP_100::setdpi()
{
outcode("\x1b\x2a\x74\x37\x35\x52");
}
```

```
/*  设置75dpi的图象分辨率  */
void HP_75::setdpi()
{
outcode("\x1b\x2a\x74\x33\x30\x30\x52");
}

/*  换页  */
void EPSON_PRINTER::outpaper(void)
{
biosprint(0,0x0c,0);
}

/*  打印机初始化  */
void EPSON_PRINTER::init()
{
outcode("\x1b\x40\x1b\x3c");
}

/*  设置单向打印  */
void EPSON_PRINTER::setdx()
{
outcode("\x1b\x55\x1");
}

/*  设置n/180英寸行距  */
void EPSON_PRINTER::setjg180(unsigned char c180)
{
biosprint(0,0x1b,0);
biosprint(0,0x33,0);
biosprint(0,c180,0);
}

/*  EPSON打印机图象打印，所有图象模式共用此函数  */
void EPSON_PRINTER::print()
{
unchar **data,*buf,c1,c2;
int yi,i,j,yn,xn,y,bn,K1,K2;
int rowd=dpi*2/15;  //每列点数
int rowb=dpi/60;  //每列字节数
byten=vga->WIDE/8;  //每行象素所用字节数
init();  //初始化打印机
setjg180(24);  //设置24/180英寸行距
setdx();  //设置单向打印
xn=vga->WIDE;  //图象列数
yn=(vga->HIGH+rowd-1)/rowd;  //图象的打印头行数
data=(unchar **)calloc(rowb,sizeof(void *));
//存放经过转换后的一个打印头行的图象数据
for(i=0;i<rowb;i++)
    data[i]=(unchar *)calloc(xn,1);
y=0;
for(yi=0;yi<yn;yi++)
    {
```

```
    c1=0x80;
    bn=K1=0;
    for(i=0;i<rowb;i++)
       for(j=0;j<xn;j++)
           data[i][j]=0;
    /*  以下循环，将缓存区中的数据转换为符合打印格式的数据，结果放在data中  */
    for(i=0;i<rowd;i++)
        {
        buf=getprtdata(y);
        y++;
        if(buf==0)
            breaK;
            c2=*buf;
            K2=0;
            for(j=0;j<xn;j++)
                {
                if( c2&0x80 )
                    data[bn][j] |= c1;
                K2++;
                c2 <<=1;
                if( K2>=8 )
                    {
                    buf++;
                    c2=*buf;
                    K2=0;
                    }
                }
            K1++;
            c1 >>= 1;
            if( K1>=8 )
                {
                c1=0x80;
                bn++;
                K1=0;
                }
        }
    setmode(xn);  //设置图象模式
    for(i=0;i<xn;i++)
        for(j=0;j<rowb;j++)
            biosprint(0,data[j][i],0);  //输出图象数据
    nextrow();  //换行、回车
    }
setjg180(30);  //将行距还原为30/180英寸
for(i=0;i<rowb;i++)
    free(data[i]);
free(data);
}

/*************************************************************************
功能：输出图象模式设置指令
输入参数：pxn＝图象列数
          code＝具体图象模式的图象设置指令头
```

```
返回值：无
*************************************************************************/
void EPSON_PRINTER::setmodeb(int pxn,unchar *code)
{
int i;
i=pxn>>8;
code[4]=i;
i=pxn&0xff;
code[3]=i;
for(i=0;i<5;i++)
    biosprint(0,code[i],0);
}

/*  360×360图象模式设置  */
void EPSON_360::setmode(int pxn)
{
unsigned char code[6]={0x1b,0x2a,72,0,0,0};
setmodeb(pxn,code);
}

/*  180×180图象模式设置  */
void EPSON_180::setmode(int pxn)
{
unsigned char code[6]={0x1b,0x2a,39,0,0,0};
setmodeb(pxn,code);
}

/*  60×60图象模式设置  */
void EPSON_60::setmode(int pxn)
{
unchar code[6]={0x1b,0x2a,0,0,0,0};
setmodeb(pxn,code);
}
```

12.3 使用图形打印程序

使用图形打印程序的基本步骤如下：

①运行XMS::init()，初始化扩展内存，以能在扩展内存中建立打印缓存区；

②定义prt_buf类的一个对象A，以建立一个图象缓存区，定义时需指定图象的大小及缓存区的建立区域；

③通过对象A向缓存区输出图象；

④定义一个打印模式类对象P，定义时将对象A的指针传给P；

⑤运行P.print()，实现图形打印。

下面给出几个示例程序。

示例程序12-1　在HP打印机上以300dpi的分辨率打印一个圆。

```
void main()
{
XMS::init();
```

```
prt_buf A(2000,2000,inXMS);
A.cricle(1000,1000,800,A.setcolorto(1));
HP_300 P(&A);
P.print();
}
```

示例程序12-2　将显示在屏幕上的一个圆拷贝到打印机上。

```
void main()
{
_1024_768_16 A;
A.init();
A.cricle(512,384,383,A.setcolorto(4));
HP_300 P(&A);
P.print();
A.close();
}
```

示例程序12-3　在宽行打印纸上打印出一个40×40厘米的空心矢量汉字。

```
void main()
{
XMS::init();
prt_buf A(960,960,inMEM);
A.setcolor(1);
HZUCK T("C:\\UCDOS\\HZKSLKTJ",inHD,&A);  //&A将字符输出导向打印缓存区
T.setsize(960,960,0);  //设置字符大小
T.sethollow(1);  //设置空心字符
T.outtextxy(0,0,"楷");
ESPON_60 P(&A);
P.print();
}
```

第 13 章　鼠 标 驱 动

图形显示几乎总是和鼠标联系在一起的，各种图形软件大都将鼠标作为与用户交互的主要设备，这一章将介绍与本书的图形显示程序相关联的鼠标驱动技术及通用的鼠标编程技术，并实现一组鼠标操作的基础支持程序。

13.1　概　　述

每个鼠标都带有一个鼠标驱动程序，鼠标驱动程序以设备驱动程序或TRS的形式安装在系统中，鼠标驱动程序负责完成如下3项工作：一是在屏幕上显示鼠标光标并维持光标的移动；二是向应用程序提供鼠标的状态，这包括鼠标光标在屏幕上的位置及鼠标的各个按钮是被按下还是放开；三是为应用程序提供一些鼠标操作的辅助功能，如隐藏光标、设置移动速度、改变光标形状等，鼠标驱动程序以中断调用的形式向应用程序提供所有的鼠标操作功能。

尽管已存在鼠标驱动程序，但应用程序在使用鼠标时仍有两个问题需要解决，一是，鼠标中断功能只能向应用程序报告鼠标的静态状态，而应用程序需要获取的则是一些动态的鼠标事件，如单击、双击、移动、持续按下等，这就需要专门编写程序来跟踪鼠标状态的变化过程，以生成动态的鼠标事件；二是，鼠标驱动程序只能对640×480×16色、320×200×256色这两种标准的VGA图形模式提供支持，也就是说，在那些扩展的VGA图形模式下，鼠标驱动程序无法显示鼠标光标，这时就需要专门编写程序来进行光标的显示并维持光标的移动。下面首先介绍鼠标驱动程序所提供的中断功能，然后介绍上面两方面问题的解决方法及相应的程序。

13.2　鼠标中断功能

安装了鼠标驱动程序之后，系统中将出现一组鼠标操作的中断功能，这组功能的中断向量号为33H。不同厂商或不同版本的鼠标驱动程序所提供的中断功能都存在一定的差别，但它们所提供的基本功能都是相同的，下面主要介绍那些为各种鼠标驱动程序所共同支持的那些基本鼠标操作中断功能。

功能0：初始化鼠标

输入参数：AX＝0

返回值：　AX＝0　　未安装鼠标驱动程序

　　AX＝-1　安装了鼠标驱动程序

　　BX＝按钮数

该功能将初始化鼠标状态，将有关参数设置为缺省值。其所设置的鼠标状态如下：

显示页	0页
光标活动范围	整个屏幕(对VGA扩展图形模式无效)
光标位置	屏幕中央(对VGA扩展图形模式无效)
光标状态	隐藏
光标形状	箭头(对VGA扩展图形模式无效)

用户中断　　无
Mickey/Pixel值　　水平方向＝8:8；垂直方向＝16:8
倍速阀值　　每秒64个mickey

mickey为鼠标移动单位，1个mickey约为0.02in，0.5mm。

功能1：显示鼠标光标

输入参数：AX＝1
返回值：无

驱动程序维持着一个小于等于零的光标显示标志，当该标志为0时，光标就将显示在屏幕上，当该标志小于0时，光标就不显示。本功能仅仅是将光标显示标志加1，因此并不一定能打开光标显示。当标志已为0时，该功能将不起作用。用功能0进行鼠标初始化之后，光标显示标志为-1。

功能2：隐藏鼠标光标

输入参数：AX＝2
返回值：无

本功能将光标显示标志减1。仅执行本功能一次，就能将光标真正隐藏起来。如果多次执行了本功能，就需多次调用功能1才能使光标显示出来。

功能1和功能2是应用程序中常用的两项功能，当应用程序要进行屏幕操作时，应在操作之前调用功能2将光标关闭，操作完后再调用功能1将光标打开。否则，如果操作的区域与光标重合，就会发生错误。

功能3：取鼠标状态

输入参数：AX＝3
返回值：BX＝按钮状态
　　CX＝X坐标
　　DX＝Y坐标

BX中的一位对应于一个鼠标按钮，位0对应于左按钮，位1对应于右按钮，位2对应于中按钮，位3～位7无定义。当某位为1时，表示相应按钮处于按下状态，为0时表示处于放开状态。

这是应用程序最常用的一项功能。

功能4：置鼠标位置

输入参数：AX＝4
　　CX＝新位置的X坐标值
　　DX＝新位置的Y坐标值
返回值：无

若所给的位置超出了屏幕的范围或超出了功能7和功能8所设置的范围，它会调整到最近的位置上。

功能5：取按钮按下信息

输入参数：AX＝5
　　BX＝所要查询的按钮
　　0　左按钮

1 右按钮
2 中按钮

返回值：AX＝所有按钮的目前状态，其定义同功能4
BX＝所查询的按钮自上次调用功能以来按下的次数
CX＝所查询的按钮最后一次按下时的光标水平位置
DX＝所查询的按钮最后一次按下时的光标垂直位置

功能6：取按钮放开信息

输入参数：AX＝6
BX＝所要查询的按钮，其定义同功能5
返回值：AX＝所有按钮的目前状态，其定义同功能4
BX＝所查询的按钮自上次调用本功能以来放开的次数
CX＝所查询的按钮最后一次放开时光标的水平位置
DX＝所查询的按钮最后一次放开时光标的垂直位置

功能7：置鼠标的X界限

输入参数：AX＝7
CX＝X的最小值
DX＝X的最大值
返回值：无

若CX的值大于DX，两者的值将交换。执行本功能后，若光标在界限之外，其将被移到最靠近原位置的边界上。

功能8：置鼠标的Y界限

输入参数：AX＝8
CX＝Y的最小值
DX＝Y的最大值

若CX的值大于DX，两者的值将交换。执行本功能后，若光标在界限之外，其将被移到最靠近原位置的边界上。

功能9：设置图形光标的形状

输入参数：AX＝9
BX＝热点位置的X值(-16到16)
CX＝热点位置的Y值(-16到16)
ES:DX＝光标图象数据的地址
返回值：无

该功能所设置的光标形状对VGA的扩展图形模式无效，但热点设置是有效的。初始化之后，图形模式下的光标形状为箭头，热点位置为(0,0)。

功能0Ah：设置文本光标类型

功能0Bh：读移动量

输入参数：AX＝0Bh
返回值：CX＝水平移动的mickey数
DX＝垂直移动的mickey数

所返回数的为有符号的整型数，负数表示向左、向上的移动，正数表示向右、向下的移动。所返回的为两次调用本功能之间的移动量。

功能0Ch：安装用户的事件处理程序

输入参数：AX＝0Ch

CX＝调用掩码，其每一位对应一个事件，表13-1给出了其定义

ES:DX＝用户事件处理程序的地址

返回值：无

调用掩码指明了引发用户事件处理程序的条件，当某一鼠标事件发生且调用掩码中的相应位为1时，ES:DS所指向的程序就将被执行，此时CPU各寄存器的值如表13-2所示，DS将指向鼠标驱动程序的数据段，如用户程序需取自己的数据，则需将DS指向自己的数据段。

表13-1　调 用 掩 码

位	鼠标事件	位	鼠标事件
0	鼠标移动了	4	右按钮放开
1	左按钮按下	5	中按钮按下
2	左按钮放开	6	中按钮放开
3	右按钮按下		

表13-2　调用用户程序时寄存器的状态

寄存器	内容	寄存器	内容
AX	事件引发位(定义同表13-1，但只有引发这一事件的位存在)	DX	光标垂直位置
BX	按钮状态(同功能3)	DI	水平方向移动量(同功能0Bh)
CX	光标水平位置	SI	垂直方向移动量(同功能0Bh)

初始化时，调用掩码为0。若程序使用了本功能，就应在程序退出前用功能0、本功能或功能14h来将调用掩码恢复为0。

功能0Dh：打开光笔模拟

功能0Eh：关闭光笔模拟

功能0Fh：设置鼠标速度

输入参数：AX＝0Fh

CX＝水平位置变化8个象素时所需的mickey数(缺省值为8)

DX＝垂直位置变化8个象素时所需的mickey数(缺省值为16)

返回值：无

CX和DX的最高为必须为0。最小值为1，值越低鼠标在屏幕上移动越快。

功能10h：条件光标关闭

输入参数：AX＝10h

CX＝区域左上角的X坐标值

DX＝区域左上角的Y坐标值
SI＝区域右下角的X坐标值
DI＝区域右下角的Y坐标值

返回值：无

当光标移到所给定的区域内时，光标将自动关闭，但再移出该区域时，光标并不能自动打开，而需用功能1来打开光标。

功能13h：设置倍速阀值

输入参数：AX＝13h
DX＝倍速阀值(mickey/s)

返回值：无

当鼠标以等于或大于倍速阀值的速度移动时，屏幕上光标的移动速度将加倍。

功能14h：交换用户事件处理程序

输入参数：AX＝14h
CX＝新的调用掩码
ES:DX＝新的用户事件处理程序的指针

返回值：CX＝已有的调用掩码
ES:DX＝已有的用户程序的指针

该功能类似于功能0Ch，不同的是，本功能可以获得已有的事件处理程序的调用掩码和指针，以用于恢复。

功能15h：取保存状态缓冲区的大小

输入参数：AX＝15h

返回值：BX＝存放鼠标状态所需缓冲区的大小，以字节计

这一功能为功能16h作准备。

功能16h：保存鼠标状态

输入参数：AX＝16h
ES:DX＝保存鼠标状态的缓冲区的指针

返回值：无

本功能将鼠标的当前状态保存到ES:DX所指的缓冲区中，在调用本功能前需用功能15h获得所需缓冲区的大小，并分配一个所需大小的缓冲区。

当挂起一个使用鼠标的程序而运行另外一个使用鼠标的程序时，就应使用本功能保存鼠标的状态，在第二个程序结束又回到第一个程序时，再使用功能17h恢复鼠标状态。

功能17h：恢复鼠标状态

输入参数：AX＝17h
ES:DX＝保存有鼠标状态的缓冲区的指针

返回值：无

本功能用于恢复由功能16h所保存的鼠标状态。

13.3 扩展图形模式下鼠标光标的维持

在VGA的各种扩展图形模式下，鼠标驱动程序无法显示和维持屏幕上的鼠标光标，这

里给出一种在扩展图形模式下维持鼠标光标的方法，并实现一组相应的程序。

13.3.1　维持鼠标光标的方法

在各种图形模式下维持鼠标光标的基本方式为：当由功能1打开鼠标光标，第一次在屏幕上显示光标时，首先保存光标所要覆盖的屏幕矩形块，然后将光标显示到屏幕上；当鼠标移动，光标要显示到一个新的位置时，则首先在原处恢复上次所保存的矩形块，再保存新位置的矩形块，然后在新位置显示出光标；当调用功能2隐藏光标时，则在原位置恢复上次所保存的矩形块。按照这种维持方式，在编程中就主要有两个问题需要解决，一是如何显示鼠标光标，二是如何维持光标的移动。至于光标处背景的保存和恢复，则可直接用图形显示程序所提供的各种图形模式下的块保存和块恢复功能来完成。

鼠标驱动程序给鼠标光标定义了一种专门的图象数据格式，一个光标图象数据由两个16×16点阵的图象掩码共64个字节所组成，将第1个图象掩码以与方式写到屏幕上，然后将第2个掩码以异或方式写到屏幕上，即完成鼠标光标的显示，这样显示出的鼠标光标只能包含4种颜色：黑色、白色、原背景色、原背景色的反色，其中有意义的只有黑、白两色。这种光标显示方式主要适合于640×480×16色这种图形模式，在其它的色彩模式下，这种方式所能显示的颜色就显得太少，而在高分辨率模式下16×16点阵的光标则显得太小，而且在其它的色彩模式下，这种显示方式就不再能获得硬件的支持。因而在各种扩展图形模式下就不再适合采用鼠标驱动程序所使用的光标显示方式。程序中采用了另外一种更具有C++特点的方式来进行鼠标光标的显示，即用一个类来实现光标显示，在该类中即包含了光标的图象数据，也包含了对图象数据的操作，这时的光标形状就不必再受一定的图象格式的限制，而可以很灵活地实现各种大小、各种颜色的光标显示，可以很方便地针对不同的色彩模式定义相应的多种光标显示类。

另一个需要解决的问题就是，如何维持鼠标光标在屏幕上的持续移动，可采用的方法有三种：一是，利用主机系统每秒18.2次的时钟中断，定时检测鼠标的位置，如有移动就在新的位置显示鼠标光标；二是，用鼠标中断功能0Ch，安装一个对鼠标移动事件作出响应的事件处理程序，由该程序维持光标的移动；三是，当应用程序要使用鼠标时，就必须对鼠标事件进行持续的循环查询，这就可在鼠标事件获取程序中来维持鼠标光标的移动。前两种方式都需要以中断函数的形式来实现，而定义在类中的中断函数必须是静态的，这对光标显示类的结构有较大的影响，因此程序中采用第三种方法，即在鼠标事件获取程序中实现维持光标移动的功能，这种方式不仅能避免中断函数的使用，而且能给光标显示提供更大的灵活性，这种方式也有一个缺点，就是当应用程序不需要获取鼠标事件而暂时停止对鼠标事件的查询时，这时鼠标光标就将静止不动，但这对程序的运行没有影响，而且可以避免光标移动对CPU时间的占用。

下面将首先实现鼠标光标显示程序，然后在鼠标驱动接口程序中实现维持鼠标光标移动的功能。

13.3.2　光标显示基础程序

这里定义了光标显示的一个基类，在该类中实现了那些维持光标显示的基本功能，但该类并不能显示一个具体的光标，具体的光标显示类由应用程序根据其需要来从该类派生得到，下面给出该类的说明

系统程序13-1 VGAMOUSE.H

```
/*  鼠标光标显示基类 */
class MouseMap {
    friend class MOUSE;  //MOUSE类将在后面定义，该类要用到MOUSE类中的一些私有成分
public:
    MouseMap(VGABASE *v) { vga=v; }
      //构造函数，连接一种图形模式类，以使用其图形显示功能
    ~MouseMap();  //
    void setup(void);  //将一个具体的光标显示类对象的指针传给MOUSE类
protected:
    VGABASE *vga;  //图形模式类对象的指针
    void *buf;  //保存背景区域的缓冲区的指针
    int wide,high;  //光标的宽度和高度，以象素点计
    int centre_X,centre_Y;  //光标热点的位置
    int x1,y1;  //所保存的背景区域左上角的位置
    unsigned activMask;  //激发active()函数的事件掩码

    void putback(void);  //恢复背景区域
    void show(int x0,int y0);  //显示光标
    virtual void active(Event evt) { }  //用户事件处理函数
    virtual void init(void) { }  //鼠标光标初始化
    virtual void draw(int x0,int y0)=0;  //画鼠标光标
    } ;
```

MOUSE类是与本类相关联的一个类，该类将在下一节中定义，鼠标光标的移动、隐藏、打开将在MOUSE类中实现，因此MOUSE类必须获得一个光标显示类对象的指针。

MouseMap::draw()是一个纯虚函数，该函数必须在具体的光标显示类中定义，最终的光标显示将由该函数来完成，另两个虚函数MouseMap::active()和MouseMap::init()用于辅助光标显示，以实现更生动、更复杂的光标显示，它们可以缺省。

下面给出MouseMap类的定义。

系统程序12-2 MOUSE.CPP

```
/*  析构函数。*/
MouseMap::~MouseMap()
{
MOUSE::hide();
free(buf);  //释放保存背景的缓冲区
MOUSE::mmap=0;
  //将所要删除的对象的指针从MOUSE类中清除，以免MOUSE类使用一个已不存在的对象
MOUSE::show();
}

/*  设置光标热点、设置屏幕范围、分配保存背景的缓冲区、将光标显示类对象的指针传给MOUSE类，以使其能维持光标的移动、隐藏、打开  */
void MouseMap::setup()
{
```

```
MOUSE::centre_X=centre_X;  //设置光标热点
MOUSE::centre_Y=centre_Y;  //设置光标热点
MOUSE::max_X=vga->WIDE-wide-1;
MOUSE::max_Y=vga->HIGH-high-1;
MOUSE::restoreregion();  //设置屏幕范围
MOUSE::hide();
size_t size;
size=(size_t)vga->imagesize(0,0,wide,high);
buf=malloc(size);  //分配缓冲区
init();
MOUSE::mmap=this;  //连接到MOUSE类
MOUSE::show();
}

/*  恢复光标的背景区域  */
void MouseMap::putback(void)
{
vga->putimageMEM(x1,y1,wide+1,high+1,buf);
}

/*  显示光标  */
void MouseMap::show(int x,int y)
{
COLOR col=vga->CUR_COLOR;  //保存当前颜色
int fst=vga->getfillstyle();  //保存填充模式
x1=x; y1=y;  //保存光标位置
vga->getimageMEM(x1,y1,x1+wide,y1+high,buf);  //保存背景
draw(x,y);  //画出光标
/*  恢复当前颜色和填充模式。因draw()有可能改变这两项内容  */
vga->setfillstyle(fst);
vga->CUR_COLOR=col;
}
```

13.3.3　各种色彩模式下的光标显示程序

一个具体的光标显示类至少要定义两个函数：构造函数和画光标函数，构造函数必须完成三项工作：给出光标的宽度和高度；设置光标热点的位置；运行setup()函数，以将其自身连接到MOUSE类，以实现对鼠标光标的维持。对画光标函数所画光标的宽度及高度必须与构造函数所给的一致。

这里对每种色彩模式定义了一个光标显示类，以为每种色彩模式提供一种鼠标光标。所实现的程序虽然都是系统程序，但都带有一定的示例性，以介绍在应用程序中如何从MouseMap类中派生光标显示类。

1. 16色模式

这里实现了一个在16色模式下显示一个16×16点阵大小的箭头光标的类，该类也可用于256色模式，在1024×768分辨率下，光标则显得太小。该类只定义了两个最基本的函数。

系统程序12-3 VGAMOUSE.H

```
class MouseMap16 : public MouseMap
    {
public:
    MouseMap16(VGABASE *v);  //构造函数
protected:
    void draw(int x0,int y0);  //画光标
    } ;
```

系统程序13-4 MOUSE16.CPP

```
MouseMap16::MouseMap16(VGABASE *v) : MouseMap(v)
{
wide=10;  //光标的宽度
high=15;  //光标的高度
centre_X=0;  //热点的横坐标
centre_Y=0;  //热点的纵坐标
setup();  //将该光标显示类连接到MOUSE类
}

/*************************************************************************
功能：画鼠标光标
输入参数：x0＝鼠标光标左上角的横坐标值
          y0＝鼠标光标左上角的纵坐标值
返回值：无
*************************************************************************/
void MouseMap16::draw(int x0,int y0)
{
int i;
vga->setcolor(15);
for(i=2;i<10;i++)
           vga->scanline(x0+1,x0+i,y0+i);
vga->scanline(x0+1,x0+5,y0+10);
vga->putpixel(x0+1,y0+1);
vga->putpixel(x0+1,y0+11);
vga->line(x0+5,y0+11,x0+7,y0+15);
vga->line(x0+6,y0+11,x0+8,y0+15);
vga->setcolor(0);
vga->putpixel(x0+1,y0+12);
vga->putpixel(x0+2,y0+11);
vga->line(x0,y0,x0,y0+12);
vga->line(x0+1,y0,x0+10,y0+9);
vga->line(x0+4,y0+11,x0+6,y0+15);
vga->line(x0+6,y0+10,x0+9,y0+15);
}
```

2. 256色模式

在该模式下实现了一个具有动态效果的光标显示类，其active()函数通过循环改变8个颜色索引值的DAC寄存器值来实现动态效果。因DAC寄存器操作功能不是VGABASE类的功能，因此在该类中定义了一个VGA256类的指针，以获得DAC操作功能。

系统程序13-5　VGAMOUSE.H

```
--------------------------------------------------------------------------
class MouseMap256 : public MouseMap
    {
public:
    MouseMap256(VGABASE *v);
protected:
    class VGA256 *v256;
    void init(void);
    void active(Event evt);
    void draw(int x0,int y0);
    } ;
--------------------------------------------------------------------------
```

系统程序13-6　MOUSE256.CPP

```
--------------------------------------------------------------------------
MouseMap256::MouseMap256(VGABASE *v) : MouseMap(v)
{
wide=19;
high=23;
centre_X=0;
centre_Y=0;
activMask=evMouse;
  //设置引发active()的掩码，evMouse为将在后面给出的一个宏，表示所有的鼠标事件
v256=(VGA256 *)v;  //将VGABASE类的指针赋给v256，以获得DAC操作功能
setup();
}

void MouseMap256::init(void)
{
Event ev;
active(ev);  //将所用到的8个颜色索引值初始化
}

/*  循环设置DAC寄存器的值，以产生动态效果  */
void MouseMap256::active(Event evt)
{
static unchar rgb[8]={0x60,0x70,0x80,0x90,0xa7,0xbf,0xdf,0xff};
static k=0;
int i,j;
j=k;
for(i=0;i<8;i++)
    {
    v256->setdac(0xf8+i,rgb[j],rgb[j],rgb[j]);
```

```
    j++;
    if(j>=8) j=0;
    }
k++;
if(k>=8) k=0;
}

/*  画鼠标光标  */
void MouseMap256::draw(int x0,int y0)
{
int x1=x0+1;
vga->setcolor(0xf8);
vga->putpixel(x0,y0); y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->setcolor(0xf9);
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->setcolor(0xfa);
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->setcolor(0xfb);
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->setcolor(0xfc);
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->setcolor(0xfd);
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1++; y0++;
vga->scanline(x0,x1,y0); x1=x0+11; y0++;
vga->setcolor(0xfe);
vga->scanline(x0,x1,y0); y0++;
vga->scanline(x0,x1,y0); y0++; x1=x0+12;
vga->scanline(x0,x0+1,y0);
vga->scanline(x0+6,x1,y0); y0++;
vga->setcolor(0xff);
vga->scanline(x0+6,x1,y0); y0++; x1=x0+13;
vga->scanline(x0+7,x1,y0); y0++;
vga->scanline(x0+7,x1,y0);
}
```

---

3. 真彩色模式

在该模式下实现了一个显示带阴影的十字光标的类，阴影并不是画成黑色，而是用真

彩色模式下的亮度变化函数将屏幕原色的亮度减半。

系统程序13-7　VGAMOUSE.H

```
-------------------------------------------------------------------------------
class MouseMap16M : public MouseMap
    {
public:
    MouseMap16M(VGABASE *v);
protected:
    class VGA16M *vt;
    void draw(int x0,int y0);
    } ;
-------------------------------------------------------------------------------
```

系统程序13-8　MOUSE16M.CPP

```
-------------------------------------------------------------------------------
MouseMap16M::MouseMap16M(VGABASE *v) : MouseMap(v)
{
wide=15;
high=15;
centre_X=7;
centre_Y=7;
setup();
vt=(VGA16M *)v;
}

/*  画鼠标光标  */
void MouseMap16M::draw(int x0,int y0)
{
int i,x,y;
vt->en_scanline(x0+2,x0+15,y0+9,1,2);  //画横阴影
/*  画竖阴影  */
y=y0+2; x=x0+9;
for(i=2;i<=15;i++)
    {
    vt->en_putpixel(x,y,1,2);
    y++;
    }
vga->setcolor(255,255,255);
vga->scanline(x0,x0+13,y0+7);  //画横线
vga->line(x0+7,y0,x0+7,y0+13);  //画竖线
}
-------------------------------------------------------------------------------
```

4. 高彩色模式

在该模式下实现的光标显示类，显示一个带有立体感的箭头光标。

系统程序13-9　VGAMOUSE.H

```
-------------------------------------------------------------------------------
class MouseMapHigh : public MouseMap
    {
```

```
public:
    MouseMapHigh(VGABASE *v);
    ~MouseMapHigh();
protected:
    int *xy;
    void draw(int x0,int y0);
    } ;
```

系统程序13-10 MOUSEHI.CPP

```
MouseMapHigh::MouseMapHigh(VGABASE *v) : MouseMap(v)
{
wide=14;
high=19;
centre_X=0;
centre_Y=0;
xy=(int *)calloc(10,2);  //在画光标函数中使用。该空间由析构函数来释放
setup();
}

/*  释放在构造函数中所分配的一块内存  */
MouseMapHigh::~MouseMapHigh()
{
free(xy);
}

/*  画光标  */
void MouseMapHigh::draw(int x0,int y0)
{
static int xyo[]={0,0, 0,19, 6,12, 14,14, 0,0};
int i;
for(i=0;i<10; i+=2)
    {
    xy[i]=xyo[i]+x0;
    xy[i+1]=xyo[i+1]+y0;
    }
vga->setfillstyle(0);
vga->setcolor(255,255,100);
vga->polyfill(3,xy+4);
vga->setcolor(160,160,50);
vga->polyfill(3,xy+2);
vga->poly(3,xy+2);
vga->poly(3,xy+4);
vga->setcolor(120,120,30);
vga->polyfill(3,xy);
vga->poly(3,xy);
}
```

13.4　鼠标操作接口

这里将提供一组鼠标操作接口程序，该程序主要完成以下三项功能：一是，根据鼠标状态的变化过程，向应用程序提供动态的鼠标事件；二是，在各种图形模式下，维持鼠标光标的移动，实现鼠标光标的隐藏和打开；三是，给应用程序提供一组更为方便的鼠标操作接口，使应用程序不必去直接操作鼠标中断功能。

13.4.1　事件

当一个程序采用C++编写而同时又要使用鼠标进行用户交互时，就很适合采用一种称为“事件驱动”的程序设计方法来构造程序，在事件驱动的程序设计中，程序与用户之间、程序内部的各个模块之间都采用事件来进行交互，整个程序的运行完全由事件来驱动，程序中的事件分为三类：鼠标事件、键盘事件及消息，消息是程序本身所产生的事件，它用于程序模块间的联系，这三类事件被包含在一个统一的事件结构之中。事件驱动与图形显示没有直接的关系，本书不打算对其作具体的介绍，但这里采用并实现了事件驱动程序设计中的事件定义方法及事件获取方法，鼠标事件的定义及获取被作为其中的一部分来实现。系统程序13-11给出了事件的定义，系统程序13-13给出了事件获取程序。

1. 事件定义

系统程序13-11　VGAMOUSE.H

```
--------------------------------------------------------------------------------
struct MouseEvent {  //鼠标状态
    unchar buttons;  //按钮状态
    int x,y;  //鼠标光标位置
    int keystate;  //键盘各换档键的状态
    } ;

struct MessageEvent {  //消息
    unsigned command;  //消息类型代码(由应用程序定义)
    union {  //各种格式的消息
        void far *infoPtr;
        long infoLong;
        unsigned int infoWord;
        int infoInt;
        unsigned char infoByte[2];
        char infoChar[2];
        };
    };

class Event  //事件
    {
public:
    unsigned what;  //事件类型
    union {
        int key;  //键盘事件的键码键码
        MessageEvent message;  //消息
        MouseEvent mouse;  //鼠标状态
```

```
        };
    void getevent(void);  //获取事件
    void getevent(unsigned mask);  //获取指定的事件
    void clear(void);  //清除事件
    void postmessage(void);  //发送消息
    void clearpostbox(void);  //清除消息信箱
private:
    void getMouseEvent();  //获取鼠标事件
    void getKeyEvent();  //获取键盘事件
    void getMessageEvent();  //获取消息
    };
```

MouseEvnet中所包含的只是在发生某个鼠标事件时鼠标的状态，鼠标事件的类型则定义在Enevt::what中。在应用程序中有时需要结合Shift、Ctrl及Alt这三个键的状态来处理鼠标事件，MouseEvent::keystate中存放着发生一个鼠标事件时，这三个键的状态，其每一位对应一个键，当某位为1时，表示相应的键被按下，其各位所对应的键如表13-3所示，系统程序13-12中定义了表示这三个键状态的助记符。

表13-3 MouseEvent::keystate位

位	所对应的键	位	所对应的键
0	右Shift键	2	Ctrl键
1	左Shift键	3	Alt键

程序中维持着一个消息队列，用来存放多个由应用程序所发出的消息，其维持方式与BIOS维持键盘输入队列的方式相似。Event类中的postmessage()用于将一个消息放到消息队列中，getMessageEvent()从消息队列中取出一条消息，clearpostbox()则清除消息队列。

Event::what中的每一位对应一个事件，当某位为1时，就表示发生了相应的事件，其能表示多个事件同时发生的情况。所有的鼠标事件都直接定义在Event::what中，而不是定义在MouseEvent中。其各位所对应的事件如表13-4所示，系统程序13-12中定义了各种事件的助记符。

系统程序13-12 EVTCODE.H

```
/*  事件类型(Event::what)  */
#define MouseDown      0x0001  //鼠标左按钮按下
#define MouseUp        0x0002  //鼠标左按钮放开
#define MouseAuto      0x0004  //鼠标左按钮持续按下
#define MouseDouble    0x0008  //鼠标左按钮双击
#define MouseDownR     0x0010  //鼠标右按钮按下
#define MouseUpR       0x0020  //鼠标右按钮放开
#define MouseAutoR     0x0040  //鼠标右按钮持续按下
#define MouseDoubleR   0x0080  //鼠标右按钮双击
#define MouseDownM     0x0100  //鼠标中按钮按下
#define MouseUpM       0x0200  //鼠标中按钮放开
#define MouseAutoM     0x0400  //鼠标中按钮持续按下
#define MouseDoubleM   0x0800  //鼠标中按钮双击
```

```
#define MouseMove      0x1000  //鼠标移动
#define evMouse        0x1fff  //鼠标事件
#define evButton       0x0fff  //鼠标按钮事件
#define evKey          0x2000  //键盘事件
#define evMessage      0x4000  //消息
#define evNothing      0x0000  //空事件
/*  鼠标按钮状态(MouseEvent::Buttons)  */
#define mbLeftButton   0x01  //左按钮按下
#define mbRightButton  0x02  //右按钮按下
#define mbMiddleButton 0x04  //中按钮按下
/*  Shift、Ctrl、Alt三键的状态  */
#define RightShiftDown  0x01  //右Shift键按下
#define LeftShiftDown   0x02  //左Shift键按下
#define ShiftDown       0x03  //Shift键按下
#define CtrlDown        0x04  //Ctrl键按下
#define AltDown         0x08  //Alt键按下
/*  键盘按键码(Event::key)  */
#define DEL            339  //Delete键
#define INS            338  //Insert键
#define HOME           327  //Home
#define END            335  //End
#define PGUP           329  //PgUp
#define PGDN           337  //PgDn
#define _UP            328  //上箭头键
#define _DOWN          336  //下箭头键
#define LEFT           331  //左箭头键
#define RIGHT          333  //右箭头键

#define _F1            315
#define _F2            316
#define _F3            317
#define _F4            318
#define _F5            319
#define _F6            320
#define _F7            321
#define _F8            322
#define _F9            323
#define _F10           324

#define BACKUP         8
#define TAB            9
#define SHIFTTAB       271
#define ESC            27
#define ENTER          13

#define CTRLEND        373
#define CTRLHOME       375
#define CTRLPGDN       374
#define CTRLPGUP       132
```

```
#define CTRLRIGHT      371
#define CTRLLEFT       132
```

表13-4 Event::what位

位	事 件	位	事 件
0	鼠标左按钮按下	8	鼠标中按钮按下
1	鼠标左按钮放开	9	鼠标中按钮放开
2	鼠标左按钮持续按下	10	鼠标中按钮持续按下
3	鼠标左按钮双击	11	鼠标中按钮双击
4	鼠标右按钮按下	12	鼠标移动
5	鼠标右按钮放开	13	键盘事件
6	鼠标右按钮持续按下	14	消息
7	鼠标右按钮双击		

表13-4中，各鼠标按钮的按下和放开，指的是动作而不是状态。

2. 事件获取

这里给出获取各类事件的程序。

系统程序13-13 MOUSE.CPP

```
/*  获取键盘事件。实现该函数的主要问题就是，在没有键盘输入时不要等待，而应立即返回，如等待按键就无法获取鼠标事件  */
void Event::getKeyEvent()
{
int k,a,b;
if(bioskey(1)==0)  //如果没有键盘输入
    return;  //返回
k=bioskey(0);
a=k>>8;
b=k&0x00ff;
if(b==0)  //如不为ASCII码键
    k=256+a;
else  //如为ASCII码键
    k=b;
what=evKey;
key=k;
}

/*  获取鼠标事件。具体由MOUSE::getevent()实现  */
void Event::getMouseEvent()
{
MOUSE::getevent( *this );
}

struct PostBox {  //消息队列
    static Event post[8];  //可存放8个消息
```

```
    static int beg;  //首指针
    static int num,maxNum;  //当前存放数及最大可存放数
    } ;
Event PostBox::post[8];
int PostBox::beg=0;
int PostBox::maxNum=8;
int PostBox::num=0;

/*  获取消息  */
void Event::getMessageEvent()
{
if( PostBox::num==0 )
    what = evNothing;
else
    {
    *this=PostBox::post[PostBox::beg];
    PostBox::beg++;
    PostBox::num--;
    if( PostBox::beg>=PostBox::maxNum )
        PostBox::beg = 0;
    }
}

/*  发出一条消息  */
void Event::postmessage(void)
{
if(PostBox::num<PostBox::maxNum)
    {
    int i=PostBox::beg+PostBox::num;
    if( i>=PostBox::maxNum )
        i -= PostBox::maxNum;
    PostBox::post[i]=*this;
    PostBox::num++;
    }
}

/*  清除消息队列  */
void Event::clearpostbox()
{
PostBox::beg=PostBox::num=0;
}

/*  获取事件  */
void Event::getevent()
{
what=evNothing;
getMessageEvent();
if( what == evNothing )
    {
```

```
    if( MOUSE::OK )
        getMouseEvent();
    if( what == evNothing )
        getKeyEvent();
    }
}

/*  获取一指定类型的事件  */
void Event::getevent(unsigned mask)
{
if(mask==0)
    return;
do {
    getevent();
} while( (what&mask)==0 );
}

/*  清除事件  */
void Event::clear(void)
{
what=evNothing;
}
```

13.4.2 鼠标操作接口程序

程序中的每项鼠标操作功能用一个函数来实现，所有的函数都包含在一个类MOUSE中，各函数都被说明为静态函数，因此不需要定义具体的对象即可直接使用它们。程序只实现了那些常用的鼠标操作功能，如应用程序需要使用其它的功能，可以通过直接调用鼠标中断功能来实现，本程序提供了两个调用鼠标中断功能的接口函数。下面给出程序的说明及定义。

系统程序13-14 VGAMOUSE.CPP

```
class MOUSE {
    friend class MouseMap;  //其在维持鼠标光标时要用到MouseMap类的一些私有成分
public:
    static int OK;  //鼠标驱动程序是否可用的标志
    static void getevent( Event& );  //获取鼠标事件
    static int init(void);  //鼠标初始化
    static void hide(void);  //隐藏光标
    static void show(void);  //打开光标
    static void getxy(int &x,int &y);  //获得鼠标光标的位置
    static void putxy(int x,int y);  //设置鼠标光标的位置
    static void putcentre(void);  //将鼠标位置设置到屏幕中央
    static int inarea(int left,int top,int right,int bottom);
      //判别鼠标光标是否处于某一区域之内
    static void setregion(int left,int top,int right,int bottom);
     //设置鼠标光标的活动范围
```

```
    static void restoreregion(void);  //将鼠标光标的活动范围设置为整个屏幕
    static void setspeed(int n);  //设置光标移动速度
    static void setdoubletime(int n) { doubleInterval=n; }  //设置双击事件的间隔
    static void setautotime(int n) { autoInterval=n; }
      //设置形成按钮持续按下事件的延迟时间
    static void mouse_intr(int&,int&,int&,int&);  //鼠标中断调用接口
    static void mouse_intr(unsigned,unsigned,unsigned,unsigned,unsigned);
       //鼠标中断调用接口
private:
    static class MouseMap *mmap;  //光标显示类对象的指针
    static int showState;  //光标隐藏/打开计数值
    static int max_X,max_Y;  //屏幕范围
    static int centre_X,centre_Y;  //光标热点位置
    static long lastDownTime,lastDownTimeR,lastDownTimeM;  //上次按下各按钮的时间
    static int doubleInterval,autoInterval;  //双击间隔、持续按下时间的延迟时间

    static void getstate( struct MouseState& );  //获取鼠标状态
} ;
```

系统程序13-15　MOUSE.CPP

```
/*  设置各项参数的默认值  */
MouseMap *MOUSE::mmap=0;
int MOUSE::OK=0;
int MOUSE::showState=1;
long MOUSE::lastDownTime=0L;
long MOUSE::lastDownTimeR=0L;
long MOUSE::lastDownTimeM=0L;
int MOUSE::max_X=639;
int MOUSE::max_Y=479;
int MOUSE::centre_X=0;
int MOUSE::centre_Y=0;
int MOUSE::doubleInterval=7;
int MOUSE::autoInterval=7;

struct MouseState {  //鼠标状态
    unchar buttons;
    int x,y;
    long time;
    } ;
static MouseState lastMouse={0,0,0};

/*  获取鼠标事件。结果放在以引用形式输入的参数ev中  */
void MOUSE::getevent( Event& ev )
{
static int flag=1;
MouseState state;
getstate(state);  //查询鼠标状态
```

```
if(flag)  //如果是第一次读鼠标状态
    {
    flag=0;
    lastMouse=state;
    }
/*  左左按钮的事件  */
if( state.buttons&mbLeftButton )  //如果处于按下状态
    {
    if( lastMouse.buttons&mbLeftButton )  //如果上次查询也是按下状态
        {
        if( state.time-lastDownTime>autoInterval )  //如果超过一定的延迟时间
            ev.what |= MouseAuto;  //发生了持续按下事件
        }
    else  //如上次查询是放开状态
        {
        if( (state.time-lastDownTime)<doubleInterval && state.time>lastDownTime )
         //与上次按下事件的时间间隔小于一定值(后一个条件处理跨越零点的情况)
            ev.what |= MouseDouble;  //发生双击事件
        else
            ev.what |= MouseDown;  //发生按下事件
        lastDownTime=state.time;  //记录按下时间的发生时间
        }
    }
else if( lastMouse.buttons&mbLeftButton )  //如果处于放开状态且上次处于按下状态
    ev.what |= MouseUp;  //发生放开事件
/*  获取右按钮的事件  */
if( state.buttons&mbRightButton )
    {
    if( lastMouse.buttons&mbRightButton )
        {
        if( state.time-lastDownTimeR>autoInterval )
            ev.what |= MouseAutoR;
        }
    else
        {
        if( (state.time-lastDownTimeR)<doubleInterval && state.time>lastDownTimeR )
            ev.what |= MouseDoubleR;
        else
            ev.what |= MouseDownR;
        lastDownTimeR=state.time;
        }
    }
else if( lastMouse.buttons&mbRightButton )
        ev.what |= MouseUpR;
/*  获取中按钮的事件  */
if( state.buttons&mbMiddleButton )
    {
    if( lastMouse.buttons&mbMiddleButton )
        {
```

```
        if( state.time-lastDownTimeM>autoInterval )
            ev.what |= MouseAutoM;
        }
    else
        {
        if( (state.time-lastDownTimeM)<doubleInterval && state.time>lastDownTimeM )
            ev.what |= MouseDoubleM;
        else
            ev.what |= MouseDownM;
        lastDownTimeM=state.time;
        }
    }
else if( lastMouse.buttons&mbMiddleButton )
    ev.what |= MouseUpM;

if( state.x!=lastMouse.x || state.y!=lastMouse.y )  //如果鼠标位置与上次查询不同
    {
    ev.what |= MouseMove;  //发生鼠标移动事件
    if( mmap!=0 && showState==0 )  //如果存在光标显示程序且光标处于打开状态
        {
        mmap->putback();  //恢复原处的背景
        mmap->show(state.x,state.y);  //在新位置显示光标
        }
    }
ev.mouse.buttons=state.buttons;
ev.mouse.x=state.x+centre_X;
ev.mouse.y=state.y+centre_Y;
ev.mouse.keystate=bioskey(2);
lastMouse = state;
if( (mmap!=0) && ev.what&mmap->activMask )  //如果符合激发用户事件处理函数的条件
    mmap->active(ev.what);  //调用用户事件处理函数
}

/*  查询鼠标状态。结果放在输入参数ms中  */
void MOUSE::getstate( MouseState& ms )
{
int m1=0x03,m2,m3,m4;
mouse_intr(m1,m2,m3,m4);  //调用鼠标中断功能3
ms.buttons=m2;
ms.x=m3;
ms.y=m4;
ms.time=biostime(0,ms.time);
}

/*************************************************************************
功能：鼠标中断功能调用接口
输入参数：m1＝ax寄存器的值
          m2＝bx寄存器的值
          m3＝cx寄存器的值
```

```
          m4＝dx寄存器的值
返回值：无
        各寄存器的返回值放在输入参数中
*****************************************************************/
void MOUSE::mouse_intr(int &m1,int &m2,int &m3,int &m4)
{
union REGS inregs, outregs;
inregs.x.ax=m1; inregs.x.bx=m2;
inregs.x.cx=m3; inregs.x.dx=m4;
int86(0x33,&inregs,&outregs);
m1=outregs.x.ax; m2=outregs.x.bx;
m3=outregs.x.cx; m4=outregs.x.dx;
}

/*****************************************************************************
功能：鼠标中断功能调用接口
输入参数：ax＝ax寄存器的值
          bx＝bx寄存器的值
          cx＝cx寄存器的值
          dx＝dx寄存器的值
          es＝es寄存器的值
返回值：无
    与上一函数相比，本函数能多传入一个寄存器的值，但不能获得各寄存器中的返回值。
*****************************************************************/
void MOUSE::mouse_intr(unsigned ax,unsigned bx,unsigned cx,unsigned dx,unsigned es)
{
struct REGPACK reg;
reg.r_ax=ax; reg.r_bx=bx;
reg.r_cx=cx; reg.r_dx=dx;
reg.r_es=es;
intr(0x33,&reg);
}

/*  初始化鼠标，并设置鼠标是否可用的标志  */
int MOUSE::init(void)
{
int ax=0,bx,cx,dx;
mouse_intr(ax,bx,cx,dx);  //调用鼠标中断功能0
if(ax!=0) OK=1;  //可以使用鼠标
else OK=0;  //不能使用鼠标
return(OK);
}

/*  隐藏鼠标光标*/
void MOUSE::hide(void)
{
if( showState==0 && mmap!=0 )
    mmap->putback();  //用光标显示程序来实现，而不能用鼠标中断功能2来实现
showState++;
```

```
}

/*  打开光标  */
void MOUSE::show(void)
{
if( showState>0 )
    {
    showState--;
    if( showState==0 && mmap!=0 )
        {
        int m1=3,m2,x,y;
        mouse_intr(m1,m2,x,y);  //获得鼠标位置
        mmap->show(x,y);  //显示光标
        }
    }
}

/*  设置鼠标光标的位置  */
void MOUSE::putxy(int x,int y)
{
hide();
mouse_intr(4,0,x,y,0);
show();
}

/*  将鼠标光标设置到屏幕中央  */
void MOUSE::putcentre(void)
{
putxy(max_X/2-centre_X,max_Y/2-centre_Y);
}

/*  获取鼠标光标的位置  */
void MOUSE::getxy(int &x,int &y)
{
int m1=3,m2;
mouse_intr(m1,m2,x,y);
x+=centre_X;
y+=centre_Y;
}

/*  判断鼠标光标是否处于所给出的区域中，若是返回1，否则返回0  */
int MOUSE::inarea(int left,int top,int right,int bottom)
{
int x,y;
getxy(x,y);
if(x<left||x>right||y<top||y>bottom)
    return(0);
else
    return(1);
```

```
}

/*  设置鼠标光标的活动范围  */
void MOUSE::setregion(int left,int top,int right,int bottom)
{
mouse_intr(7,0,left-centre_X,right-centre_X,0);  //调用功能7，设置水平范围
mouse_intr(8,0,top-centre_Y,bottom-centre_Y,0);  //调用功能8，设置垂直范围
}

/*  将鼠标光标的活动范围设置为全屏幕  */
void MOUSE::restoreregion(void)
{
mouse_intr(7,0,0,max_X,0);
mouse_intr(8,0,0,max_Y,0);
}

/*************************************************************************
功能：设置鼠标速度
输入参数：n＝Pixel/Mickey(1～100)，该值越小速度越快，缺省值为8
返回值：无
*************************************************************************/
void MOUSE::setspeed(int n)
{
mouse_intr(0x0f,0,n,n<<1,0);  //直接调用功能0Fh
}
```

13.5 键盘模拟鼠标

由于很多PC机上都没有配备鼠标，那些使用鼠标的应用程序往往都需要同时给用户提供一套键盘的操作接口，这无疑会增加编程的负担，而且在很多情况下使用键盘往往无法完成有效的操作。这里实现了一个用键盘模拟鼠标的程序，有了该程序，在应用程序的编程中就可以不再考虑不存在鼠标的情况。

13.5.1 实现方式

一个真正有实用价值的键盘模拟鼠标程序应该满足四个方面的要求：一是，与应用程序的编程方式无关。对应用程序来说，键盘模拟鼠标的操作方式与真实鼠标的操作方式应完全相同，在应用程序的编程中不必考虑所使用的是真实鼠标还是模拟鼠标。二是，不与正常的键盘使用相冲突。使用鼠标的程序都不可能完全排除键盘的使用，如一个文字处理软件，需要用鼠标来操作菜单和控制项，但同时也要用键盘来输入字符、控制字符光标，如果鼠标模拟程序将某些键，如方向控制键、回车键、空格键给屏蔽掉了，那就无法进行正常的字符输入。三是，应具有良好的操作效果，主要是光标在屏幕上移动的速度及平滑性应与真实鼠标相近。四是，能支持那些常用的鼠标操作功能。

程序采用了如下一些处理方式来满足上述各方面的要求：

(1)将键盘模拟鼠标的各项操作功能，以与原鼠标驱动程序完全相同的格式安装到33H

号中断功能上，供应用程序调用，这样在操作方式上即可保证与真实鼠标完全一致。

(2)用打字键盘与小数字键盘之间的4个方向控制键来模拟鼠标的移动，用右Alt键来模拟鼠标的左按钮，用右Ctrl键来模拟鼠标的右按钮，用右Shift键来模拟鼠标的中按钮，所用到的这7个键在键盘上正好有两组，剩下的一组即可用作正常的键盘输入。一般的键盘读取功能无法有区别地读取这7个键，因此程序采用了9号键盘中断功能来读取这7个键。

(3)用9号键盘中断功能获取所用到的7个键的按下和放开状态，由此生成鼠标按钮的状态，并维持光标的移动。如果仅仅是获取键的击发事件，就不能完整模拟出鼠标按钮的状态，也就不能提供与真实鼠标完全一样的操作方式，同时由于键的击发率最大只能达到30次/秒，根据键的击发事件就无法维持光标快速或平滑的移动。

(4)利用应用程序对鼠标中断功能的循环调用，在鼠标中断程序中来维持鼠标光标的移动，以达到快速、平滑的光标移动效果。应用程序对鼠标中断的调用频率要远远高于每秒18.2次的系统时钟中断，利用时钟中断不可能使光标进行快速或平滑的移动。

程序是基于前面的鼠标光标显示程序及鼠标操作接口程序来实现的，因此程序中不需进行光标的显示，也不需实现光标隐藏和光标打开功能。

13.5.2　程序

所实现的键盘模拟鼠标程序由五个单独的函数组成。

系统程序13-16　KEYMOUSE.CPP

```
#include <dos.h>
void interrupt (*oldkbint0x09)(void);
void interrupt mykbint0x09(void);
void interrupt (*oldmouseintr)(void);
void interrupt mymouseintr(unsigned bp,unsigned di,unsigned si,unsigned ds,
               unsigned es,unsigned dx,unsigned cx,unsigned bx,unsigned ax);
void setup_key_mouse();
void del_key_mouse();
void move_mouse_cur();

static int KEY_EMU_MOUSE=0;  //是否安装了键盘模拟鼠标的标志
/*  安装键盘模拟鼠标驱动程序。该程序首先判别是否存在正常的鼠标驱动程序，如存在就不安装，如不存在才安装  */
void setup_key_mouse()
{
union REGS inregs, outregs;

inregs.x.ax=0;
int86(0x33,&inregs,&outregs);
if(outregs.x.ax==0)  //如果正常的鼠标驱动程序没有安装
    {
    oldkbint0x09=(void interrupt(*)())getvect(0x09);
    disable();
    setvect(0x09,(void interrupt(*)(...))mykbint0x09);  //安装INT 9
```

```
    enable();
    oldmouseintr=(void interrupt(*())getvect(0x33);
    disable();
    setvect(0x33,(void interrupt(*)(...))mymouseintr);  //安装INT 33H
    enable();
    KEY_EMU_MOUSE=1;  //已安装了模拟鼠标
    }
}

/*  删除模拟鼠标驱动程序  */
void del_key_mouse()
{
if(KEY_EMU_MOUSE)  //如果安装了模拟鼠标驱动程序
    {
    disable();
    setvect(0x09,(void interrupt(*)(...))oldkbint0x09);  //恢复原INT 9
    setvect(0x33,(void interrupt(*)(...))oldmouseintr);  //恢复原INT 33H
    enable();
    KEY_EMU_MOUSE=0;  //无模拟鼠标驱动程序
    }
}

#define MOUSE_KEY_N 7  //所使用键的数量
static unsigned char _mouse_key_value[2][MOUSE_KEY_N]= //前6个键的按下和放开的扫描码
                    {{56,29,72,80,75,77,0},{184,157,200,208,203,205,0}};
static char _mouse_key_press[MOUSE_KEY_N]={0,0,0,0,0,0,0};  //7个键的按下或放开状态
/*  9号键盘中断功能  */
void interrupt mykbint0x09()
{
static unsigned mesp,mess,invalue,i,j,k,kb,f224=0;
disable();
mesp=_SP; mess=_SS;
enable();
invalue=inportb(0x60);
if(invalue==224)
    {
    k=0;
    f224=1;
    }
else if(f224==1)
    {
    f224=0;
    k=1;
    for(i=0;i<2;i++)
        for(j=0;j<MOUSE_KEY_N-1;j++)
            if(invalue==_mouse_key_value[i][j])
                {
                _mouse_key_press[j]=1-i;  //置某个键的状态
                k=0;
```

```
                break;
                }
    }
else if(invalue==54)  //右Shift键按下
    {
    _mouse_key_press[6]=1;
    k=0;
    }
else if(invalue==182)  //右Shift键放开
    {
    _mouse_key_press[6]=0;
    k=0;
    }
else
    k=1;
disable();
_SP=mesp; _SS=mess;
enable();
if(k)
    (*oldkbint0x09)();  //执行原9号键盘中断功能
else
    {
    kb=inportb(0x61); outportb(0x61,0x80);
    outportb(0x61,kb); outportb(0x20,0x20);
    }
}

/*  模拟鼠标的有关参数  */
static int _mouse_show=-1;
static int _mouse_x=320,_mouse_y=240,_mouse_up_x,_mouse_up_y;
static int _min_x=0,_max_x=624,_min_y=0,_max_y=464;
static int _move_xy=1;
static int _delay_time,_base_delay_time=7;

/*  移动鼠标。其仅仅确定鼠标光标的位置，不进行光标显示  */
void move_mouse_cur()
{
static unsigned int f=0;
if(_mouse_key_press[2])  //如上箭头键处于按下状态
    {
    _mouse_y-=_move_xy;
    if(_mouse_y<_min_y)
        _mouse_y=_min_y;
    }
if(_mouse_key_press[3])  //如下箭头键处于按下状态
    {
    _mouse_y+=_move_xy;
    if(_mouse_y>_max_y)
        _mouse_y=_max_y;
```

```
    }
if(_mouse_key_press[4])  //如左箭头键处于按下状态
    {
    _mouse_x-=_move_xy;
    if(_mouse_x<_min_x)
        _mouse_x=_min_x;
    }
if(_mouse_key_press[5])  //如右箭头键处于按下状态
    {
    _mouse_x+=_move_xy;
    if(_mouse_x>_max_x)
        _mouse_x=_max_x;
    }
if( _mouse_x!=_mouse_up_x || _mouse_y!=_mouse_up_y )  //如光标有移动
    {
    /*  以下条件语句，进行持续移动时的加速  */
    if(f==1)  //如是持续移动
        {
        _delay_time=8;
        _move_xy=2;
        }
    else if(f>=14&&f<22)  //如持续移动超过了14次
        _delay_time--;
    else if(f==22)  //如次序移动超过了22次
        _move_xy=3;
    _mouse_up_x=_mouse_x;
    _mouse_up_y=_mouse_y;
    delay(_delay_time+_base_delay_time);
    f++;
    }
else  //光标无移动
    {
    f=0;
    _delay_time=110;
    _move_xy=1;
    }
}

/*  模拟鼠标中断功能  */
void interrupt mymouseintr(unsigned bp,unsigned di,unsigned si,unsigned ds,
              unsigned es,unsigned dx,unsigned cx,unsigned bx,unsigned ax)
{
static unsigned i;

i=bp; i=di; i=si; i=ds; i=es;
move_mouse_cur();
if(ax==0)  //初始化
    {
    ax=-1;
```

```
    bx=2;
    }
else if(ax==3)  //查询鼠标状态
    {
    cx=_mouse_x;
    dx=_mouse_y;
    bx=0;
    if(_mouse_key_press[0])
        bx=1;
    if(_mouse_key_press[1])
        bx+=2;
    if(_mouse_key_press[6])
        bx+=4
    }
else if(ax==4)  //设置光标位置
    {
    if(cx<_min_x) cx=_min_x;
    if(cx>_max_x) cx=_max_x;
    if(dx<_min_y) dx=_min_y;
    if(dx>_max_y) dx=_max_y;
    _mouse_x=cx;
    _mouse_y=dx;
    _mouse_up_x=_mouse_x;
    _mouse_up_y=_mouse_y;
    }
else if(ax==7)  //设置光标的水平活动范围
    {
    if(cx>dx)
        { i=cx; cx=dx; dx=i; }
    _min_x=cx;
    _max_x=dx;
    if(_mouse_x<_min_x)
        _mouse_x=_min_x;
    if(_mouse_x>_max_x)
        _mouse_x=_max_x;
    }
else if(ax==8)  //设置光标的垂直活动范围
    {
    if(cx>dx)
        { i=cx; cx=dx; dx=i; }
    _min_y=cx;
    _max_y=dx;
    if(_mouse_y<_min_y)
        _mouse_y=_min_y;
    if(_mouse_y>_max_y)
        _mouse_y=_max_y;
    }
else if(ax==0x0f)  //设置光标移动速度
    {
```

```
    i=cx;
    if(i<1) i=1;
    if(i>24) i=24;
    _base_delay_time=i-1;
    }
}
```

如果一个应用程序要使用键盘模拟鼠标，只需在程序开始处执行setup_key_mouse()函数，在程序退出前执行del_key_mouse()函数即可。

13.6 程序使用方式

使用前面各项鼠标程序的基本步骤如下：

①从光标显示基类MouseMap中派生出一个光标显示类，记为MouseMapX，根据需要定义该类的有关函数。如直接采用本书所提供的某个光标显示类，则不需要进行此步工作；

②如果要使用键盘模拟鼠标程序，运行setup_key_mouse()函数。该函数并不一定装载模拟鼠标驱动程序，只在系统中不存在真实鼠标时，它才装载模拟鼠标；

③定义一个图形模式类对象A；运行A.init()，设置图形模式；

④运行MOUSE::init()，初始化鼠标；

⑤定义一个光标显示类对象MouseMapX M(&A)，以建立光标显示功能；

⑥定义一个事件类对象Event evt，用于获取鼠标事件；

⑦用evt.getevent(void)或evt.getevent(unsigned)获取鼠标事件，用MOUSE类中的各个函数进行辅助的鼠标操作；

⑧如果进行了步骤②，在程序退出前运行del_key_mouse()。

在程序中使用鼠标时，需注意以下各点：

(1)必须在图形模式设置之后定义光标显示对象，否则就无法正确确定屏幕的大小。应用程序只需定义光标显示对象，而不需对它进行任何操作，对它的操作由鼠标操作接口程序来自动进行。

(2)如果应用程序要直接调用鼠标中断功能，则下面这些功能是不能使用的或是无效的：功能1、功能2、功能9、功能10h、功能15h、功能16h、功能17h。

(3)在程序中最好只用evt.getevevt()来获取各类事件，如果要另外编写程序来获取键盘事件，则应采用循环方式，而不要采用等待方式。如需要等待某类事件的发生则可使用evt.getevent(unsigned)函数。

(4)在应用程序中不要用“==”或“!=”来判别所获取事件的类型，而应用按位与操作来判别事件类型，因为有可能同时发生多个鼠标事件。

(5)如果安装了键盘模拟鼠标驱动程序，在程序退出前就一定要删除它，以恢复原来的键盘中断，否则程序退出后就无法进行键盘输入。

下面给出一个使用鼠标的示例程序，该程序定义了3个光标显示类，以显示三种不同的光标。在操作中，每个按钮对应一种光标，当按下某个按钮时，光标就变为相应的一种。

示例程序13-1

```
class MMap1 : public MouseMap
    {
public: MMap1(VGABASE *v);
protected: void draw(int x0,int y0);
    } ;

class MMap2 : public MouseMap
    {
public: MMap1(VGABASE *v);
protected: void draw(int x0,int y0);
    } ;

class MMap3 : public MouseMap
    {
public: MMap1(VGABASE *v);
protected: void draw(int x0,int y0);
    } ;

//上面三个类的定义省略

void main()
{
Event ev;
_1024_768_256 V;
MouseMap *mp;
setup_key_mouse();  //安装模拟鼠标
MOUSE::init();
V.init();
mp=new MMap1(&V);
MOUSE::putcentre();  //将鼠标光标放到屏幕中央
MOUSE::show();  //打开鼠标光标
while(1) {
    ev.getevent();
    if( ev.what&MouseDown )  //如果发生左按钮按下事件
        {
        delete mp;  //删除正在使用的鼠标光标
        mp=new MMap1(&V);  //使用第1种鼠标光标
        }
    else if( ev.what&MouseDownR )
        {
        delete mp;
        mp=new MMap2(&V);
        }
    else if( ev.what&MouseDownM )
        {
        delete mp;
        mp=new MMap3(&V);
        }
```

```
    else if( ev.what&MouseDouble )  //如果发生左按钮双击事件
        break;  //退出循环
    }
delete mp;  //去掉鼠标光标
ev.getevent(evKey);  //等待发生击键事件
V.close();
del_key_mouse();  //删除模拟鼠标
}
```

第14章　屏 幕 漫 游

屏幕漫游就是用一个较小的屏幕来观察一幅较大的图象，而将屏幕在图象上平滑移动的过程。VGA能对屏幕漫游提供硬件支持，这一章将介绍屏幕漫游的有关技术，并实现一组支持屏幕漫游的基础程序。

14.1　屏幕漫游的原理及技术

14.1.1　屏幕漫游实现原理

VGA的屏幕漫游由三项基本功能构成：虚屏定义，即在显示存储器中定义一个比实际屏幕更大的虚拟屏幕；屏幕移动，即在虚拟屏幕上移动实际屏幕；屏幕分割，即将实际屏幕分成两部分，一部分可以移动，另一部分固定不动。要实现这三项功能，实际上就是要从不同的方面来改变显示存储器与实际屏幕之间的对应关系。下面根据VGA在屏幕刷新中的数据读取方式，来看看如何在VGA上实现这三项功能。

在进行屏幕刷新时，VGA并不是固定从显示存储器的最开始处来读取第1条扫描线的数据，而是根据其所保存的一个开始地址值来确定第1条扫描线数据的开始位置。按当前模式下实际屏幕的宽度(水平象素数)来读取一条扫描线的数据，读完后，并不是在紧接着的位置来读取下一条扫描线的数据，而是根据所保存的一个偏移量值来确定下一条扫描线数据的开始位置，该偏移值表示了相邻两条扫描线数据开始位置的相对偏移。在垂直扫描过程中，每扫完一条扫描线,就对已扫完扫描线的条数进行计数，并将该计数值与其所保存的一个行比较值进行比较，如果还没有扫完整个屏幕，计数值即已等于行比较值，这时就不再从后续的显示存储器中读取数据，而是回到显示存储器的最开始处来顺序读取剩余扫描线的数据。

由上可见，在VGA中所保存的开始地址值、偏移量值及行比较值这三个参数确定了屏幕刷新中的数据读取方式，也即确定了显示存储器与屏幕之间的对应关系，修改这三个参数即可实现相应的屏幕漫游功能：增大偏移量值，即可在显示存储器中定义一个比实际屏幕大的虚拟屏幕；改变开始地址值即可实现屏幕移动；使行比较值小于屏幕高度即可实现屏幕分割。下面介绍在VGA上修改这三个参数的具体方式。

14.1.2　实现方式

与屏幕漫游有关的三个参数都存放在CRT控制器的有关寄存器中，通过寄存器操作即可修改这三个参数，CRT控制器寄存器的I/O端口及操作方式见2.1节中的有关内容。下面介绍这三个参数在寄存器中的具体保存方式。

1. 偏移量

其宽度为8位，存放在CRT控制器的索引值为13h的偏移量寄存器中。在不同的图形模式下，一个单位的偏移量值所对应的字节数和象素点数是不同的，表14-1列出了几种图形模式下偏移量的单位字节数、单位象素点数及8位的偏移量值所能表示的最大象素点数。

首先设置一种正常的图形模式，然后修改偏移量，即可定义出虚拟屏幕的宽度，所定义的宽度不能小于原屏幕的宽度，不能大于最大象素点数，且宽度必须为单位象素点数的

整数倍，在此范围内，虚屏的宽度还要受显示存储器容量的限制，在显示存储器容量一定时，所定义的虚屏越宽，其高度就越小，在定义虚屏的宽度时，必须要保证其高度不小于原屏幕的高度。虚屏的高度不需要专门定义，其大小主要取决于显示存储器容量，如在640×480×256色模式下定义了一个宽度为1024的虚屏，这时如果显示存储器的容量为512K，则虚屏的高度就不能超过512行。同时，虚屏的高度还要受开始地址值所能表示的最大象素点数的限制，这将在下面介绍。

表14-1 各种模式下的偏移量值

图形模式	单位字节数	单位象素点数	最大象素点数
640×480×16色	8	16	4080
800×600×16色	8	16	4080
1204×768×16色	4	8	2040
640×480×256色	16	16	4080
800×600×256色	16	16	4080
1024×768×256色	8	8	2040
320×200 高彩色	8	4	1020
512×480 高彩色	8	4	1020

2. 开始地址

这是一个16位的值，其高8位和低8位分别保存在两个开始地址寄存器中，这两个寄存器的索引值分别为0Ch(高8位)和0Dh(低8位)。在不同的色彩模式下一个单位的开始地址值所对应的字节数和象素点数是不同的，相应地，其所能表示的最大字节数及最大象素点数也是不同的，表14-2列出了这方面的情况。

表14-2 各种色彩模式下的开始地址值

色彩模式	单位字节数	单位象素点数	最大字节数	最大象素点数
16色	4	8	262 140	524 280
256色	8	8	524 280	524 280
高彩色	4	2	262 140	131 070

修改开始地址即可实现屏幕移动，但移动的垂直范围要受其所能表示的最大象素点数的限制，如在800×600×16色模式下定义了一个宽度为1024的虚屏，这时开始地址最大就只能移动到第511条扫描线上，加上原屏幕的高度，这时通过屏幕移动所能看到的最大虚屏高度就为1111，即使这时显示存储器容量为1M，也不可能将虚屏的高度定义为2048，这个例子即反映了开始地址值所能表示的最大象素点数对虚屏高度的限制。

3. 行比较值

其宽度为10位，其低8位放在索引值为18h的行比较寄存器中，位8放在索引值为7的溢出寄存器的位4中，位9放在索引值为9的最大扫描线寄存器的位6中。其每个单位对应于一行扫描线，其最大可表示1023行扫描线。

使行比较值小于屏幕的高度，即可将屏幕分割成上下两部分，其中上半部分屏幕的高

度等于行比较值，剩余的则为下半部分屏幕。上半部分屏幕的图象数据来自于开始地址处，其可随开始地址的改变而移动，下半部分的图象数据则总是来自于显示存储器的最开始处，在上半部分屏幕移动时，其保持不动。

屏幕分割不是屏幕漫游所必需的功能，但在某些情况下屏幕分割是比较有用的，如可以将一些提示信息、一些控制项和菜单项放在固定屏幕中。

14.1.3　对屏幕漫游的限制

上面所介绍的屏幕漫游的各项技术早在EGA上就已实现，而目前的VGA在显示存储器容量及显示模式上较EGA都已有了很大的发展，因而在目前的VGA上使用屏幕漫游技术就不可避免地会存在一些限制，下面列出所存在的一些限制。

①偏移量的取值范围限制了虚屏的宽度；

②开始地址的取值范围限制了虚屏的高度；

③在3字节的真彩色模式下不能进行屏幕漫游；

④在两种标准VGA图形模式下，虚屏只能使用一页的显示存储器，而320×200×256色模式本身已基本完全占用了一页的显示存储器，因而在这种模式下实际上就不能进行屏幕漫游；

⑤320×200的高彩色模式不能进行有效的屏幕分割。

上面列出的后两项限制可能来自于具体的显示卡，在某些型号的显示卡上可能不存在这两项限制，但又有可能出现其它的限制。

14.2　编 程 方 案

屏幕漫游的实现技术是比较简单的，但屏幕漫游是一种特殊的屏幕显示方式，如果在应用程序中直接使用屏幕漫游，那就需要应用程序也按一种特殊的方式来组织其全部的屏幕输出功能，这无疑会给应用程序的编程带来较大的麻烦。因而，编写一个屏幕漫游基础支持程序的主要问题并不在于如何实现屏幕漫游，而在于如何向应用程序提供一种简单方便的屏幕漫游使用方式，以使应用程序能以与正常的图形模式完全相同或基本相同的方式来使用屏幕漫游，而不需要对其屏幕输出功能作特殊的处理，本书所编写的程序即要达到这一目标，为达到这一目标，在编程中主要作了如下两项处理。

(1)将屏幕漫游看作是一类特殊的图形模式，称漫游模式，在一种正常的图形模式的基础上定义一个虚拟屏幕即形成一种漫游模式。在程序中对每种漫游模式定义一个类，每个漫游模式类都需要具有正常图形模式类中的所有功能，同时还要具有一些漫游功能，将各漫游模式所需要的那些漫游功能定义在一个屏幕漫游基类中，然后利用C++所支持的多重继承机制，从屏幕漫游基类和一种色彩模式类中共同派生出一个漫游模式类，以使得漫游模式类就即可获得漫游功能，也可获得图形操作、绘图、字符显示、鼠标驱动及屏幕硬拷贝等图形功能。这样应用程序就能以与正常的图形模式完全相同的方式来在漫游模式下使用各项图形功能。

(2)在程序中提供自动漫游功能，所谓自动漫游就是在基础程序中直接向最终用户提供移动屏幕的功能，而不需要应用程序来维持屏幕的移动，对应用程序来说漫游就是自动进行的。程序分别实现了键盘和鼠标的自动漫游。键盘自动漫游的实现方式与键盘模拟鼠

标类似，即用打字键盘和小数字键盘之间的4个箭头键来控制屏幕的移动，其不影响正常的键盘输入。鼠标自动漫游则利用鼠标光标显示类中的用户事件处理函数active()来实现，当发生鼠标移动事件时，即检查鼠标光标是否在实际屏幕之外，如是则移动屏幕，使鼠标光标始终处于实际屏幕之内，这样用户即可用鼠标来引导屏幕的移动。有了自动漫游功能，应用程序就可将整个虚屏看作是一个实际屏幕，而可以忽略屏幕漫游的存在，这就使得应用程序能以与正常图形模式完全相同的方式来使用漫游模式。

由于对屏幕漫游存在一些限制，程序只实现了对11种漫游模式的支持，而不是18种，每种漫游模式对应于一个漫游模式类，它们都继承于一定的色彩模式类和屏幕漫游基类，图14-1给出了类之间的继承关系，系统程序14-1给出了各类的说明。

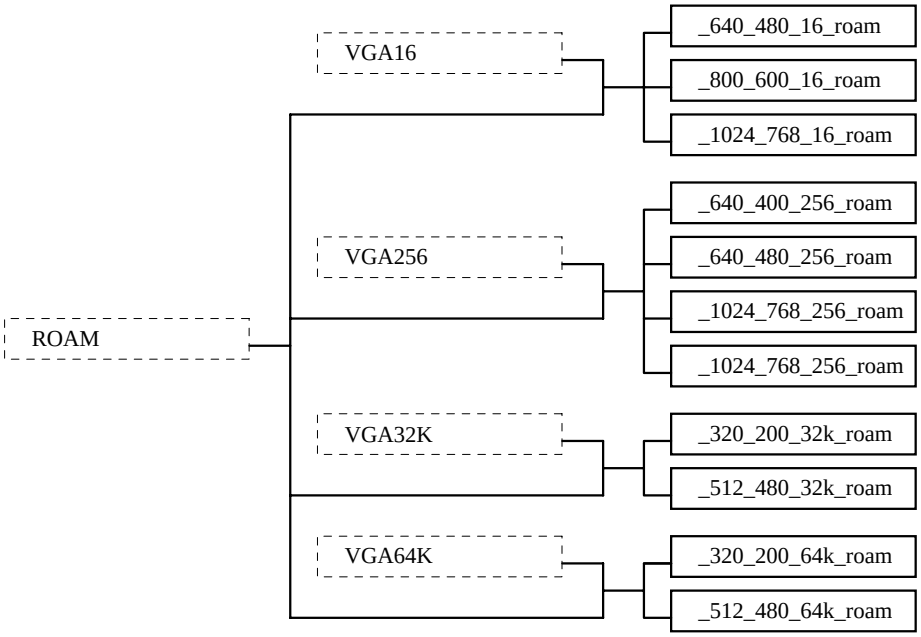

图14-1 屏幕漫游类体系

图14-1中，类ROAM为屏幕漫游基类，它是一个抽象类，最低层的11个类即为各漫游模式类。

系统程序14-1 VGAROAM

```
/*  屏幕漫游基类  */
class ROAM
    {
public:
    static int orgWide,orgHigh;  //实际屏幕的宽度和高度
    static int activHigh,fixedHigh;  //活动屏幕的高度、固定屏幕的高度
    static int begx,begy;  //在虚屏中，实际屏幕左上角的坐标
    static int endx,endy;  //在虚屏中，实际屏幕右下角的坐标

    static void roam(int dx,int dy);  //以相对值移动屏幕
    static void setbxy(int bx,int by);  //按左上角的坐标值移动屏幕
    static void setexy(int ex,int ey);  //按右下角的坐标值移动屏幕
```

```
    static void autoroam(int x0,int y0,int xn,int yn);  //自动移动屏幕
    void setup_key_roam(int d=8);  //安装键盘自动漫游功能
    void del_key_roam(void);  //删除键盘自动漫游功能
protected:
    static int minMoveY,maxMoveY;  //屏幕左上角在y方向的移动范围
    static int maxMoveX;  //屏幕左上角在x方向的移动范围
    static int minMoveWide,minSetWide;  //最小移动量、最小虚屏宽度增量
    static int VWIDE,VHIGH;  //虚屏的宽度和高度
    static int KEY_OK;  //是否安装了键盘自动漫游功能的标志
    static int dxy;  //键盘自动漫游中，每次按键移动的量

    void checkpar(void);  //检验并调整应用程序所设置的虚屏宽度和高度
    void setScreenWide(int wi);  //设置虚屏宽度
    void cutScreen(int hi);  //分割屏幕
    void setmode();  //漫游模式设置
    static void interrupt kbint0x09_roam(void);  //键盘自动漫游功能
    };

/*  640×400×256色漫游模式类  */
class _640_400_256_roam : public VGA256, public ROAM
    {
public:
    _640_400_256_roam(int vwi,int vhi,int fhi=0);  //构造函数。设置有关参数
protected:
    void setmode(void);  //模式设置
    };

class _640_480_256_roam : public VGA256, public ROAM
    {
public:
    _640_480_256_roam(int vwi,int vhi,int fhi=0);
protected:
    void setmode(void);
    };

class _800_600_256_roam : public VGA256, public ROAM
    {
public:
    _800_600_256_roam(int vwi,int vhi,int fhi=0);
protected:
    void setmode(void);
    };

class _1024_768_256_roam : public VGA256, public ROAM
    {
public:
    _1024_768_256_roam(int vwi,int vhi,int fhi=0);
protected:
    void setmode(void);
```

```
    };

class _640_480_16_roam : public VGA16, public ROAM
    {
public:
    _640_480_16_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };

class _800_600_16_roam : public VGA16, public ROAM
    {
public:
    _800_600_16_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };

class _1024_768_16_roam : public VGA16, public ROAM
    {
public:
    _1024_768_16_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };

class _320_200_32k_roam : public VGA32k, public ROAM
    {
public:
    _320_200_32k_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };

class _512_480_32k_roam : public VGA32k, public ROAM
    {
public:
    _512_480_32k_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };

class _320_200_64k_roam : public VGA64k, public ROAM
    {
public:
    _320_200_64k_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };
```

```
class _512_480_64k_roam : public VGA64k, public ROAM
    {
public:
    _512_480_64k_roam(int vwi,int vhi,int fixedhi);
protected:
    void setmode(void);
    };
```

14.3　程　　序

14.3.1　屏幕漫游基础程序

系统程序14-2　ROAM.CPP

```
#define CRT_CTL 3d4h  //CRT控制器寄存器组索引端口地址
/*  定义ROAM类中的静态数据成员  */
int ROAM::begx=0,ROAM::begy=0;
int ROAM::endx=0,ROAM::endy=0;
int ROAM::minMoveY,ROAM::maxMoveY;
int ROAM::maxMoveX;
int ROAM::minMoveWide,ROAM::minSetWide;
int ROAM::VWIDE,ROAM::VHIGH;
int ROAM::orgWide,ROAM::orgHigh;
int ROAM::activHigh,ROAM::fixedHigh;
int ROAM::KEY_OK=0;
int ROAM::dxy;

/*************************************************************************
功能：按屏幕左上角的坐标来确定屏幕在虚屏中的位置
输入参数：  ex＝屏幕左上角在虚屏中的横坐标值
            ey＝屏幕左上角在虚屏中的纵坐标值
返回值：无
    本函数及下面三个函数是程序中所提供的4个供应用程序来控制屏幕移动的函数，其中只有本函数是直接实现的，其它三个函数都通过调用本函数来实现
*************************************************************************/
void ROAM::setbxy(int bx,int by)
{
long l,l1;
unsigned seat;
if(bx<0) bx=0;  //横坐标值不能小于零
if( bx > maxMoveX )  //横坐标值太大
    bx=maxMoveX;
if(by<minMoveY)  //纵坐标值太小
    by=minMoveY;
if( by >maxMoveY )  //纵坐标值太大
    by=maxMoveY;
```

```
l1=(long)VWIDE*(long)by;
l=l1+(long)bx+(long)minMoveWide/2L;  //
l /= minMoveWide;
seat=(unsigned)l;  //计算得到开始地址值
/*  因屏幕的横向开始位置只能为开始地址单位象素点数的整数倍，因此实际设定的值和所指定的值可能会不同，下面2行，计算出实际设定的值  */
l *= (long)minMoveWide;
begx=l-l1;
begy=by;
endx=begx+orgWide-1;
endy=begy+orgHigh-fixedHigh-1;
/*  以下，将seat中的开始地址值写到两个开始地址寄存器中  */
asm {
    /*  写高8位  */
    mov bx,seat
    mov dx,CRT_CTL
    mov al,0ch
    out dx,al
    inc dx
    mov al,bh
    out dx,al
    /*  写低9位  */
    dec dx
    mov al,0dh
    out dx,al
    inc dx
    mov al,bl
    out dx,al
    }
}

/*************************************************************************
功能：移动屏幕
输入参数：  dx＝屏幕在x方向的移动量，负值向左移动，正值向右移动
            dy＝屏幕在y方向的移动量，负值向上移动，正值向下移动
返回值：无
*************************************************************************/
void ROAM::roam(int dx,int dy)
{
setbxy(begx+dx,begy+dy);
}

/*************************************************************************
功能：自动移动屏幕，以使所给的一个矩形区域处于实际屏幕之中
输入参数：  x0＝在虚屏中，矩形区域左上角的横坐标值
            y0＝在虚屏中，矩形区域左上角的纵坐标值
            xn＝矩形区域的宽度
            yn＝矩形区域的高度
返回值：无
```

```
*************************************************************************/
void ROAM::autoroam(int x0,int y0,int xn,int yn)
{
if(x0<begx)
    setbxy(x0,begy);
if(y0<begy)
    setbxy(begx,y0);
if( (x0+xn-1)>endx )
    setexy(x0+xn,endy);
if( (y0+yn-1)>endy)
    setexy(endx,y0+yn);
}

/*************************************************************************
功能：按屏幕右下角的坐标来确定屏幕在虚屏中的位置
输入参数：  ex＝屏幕右下角在虚屏中的横坐标值
            ey＝屏幕右下角在虚屏中的纵坐标值
返回值：无
*************************************************************************/
void ROAM::setexy(int ex,int ey)
{
setbxy(ex-orgWide+1,ey-orgHigh+fixedHigh+1);
}

/*************************************************************************
功能：设置虚屏宽度
输入参数：wi＝虚屏宽度
返回值：无
*************************************************************************/
void ROAM::setScreenWide(int wi)
{
wi /= minSetWide;
/*  以下，写偏移量寄存器  */
asm {
    mov dx,CRT_CTL
    mov al,13h
    out dx,al
    inc dx
    mov ax,wi
    out dx,al
    }
}

/*************************************************************************
功能：分割屏幕
输入参数：hi＝活动屏幕的高度
返回值：无
*************************************************************************/
void ROAM::cutScreen(int hi)
```

```
{
if(hi<=0||hi>=orgHigh-1)
    return;
/*  以下，将活动屏幕的高度写到有关寄存器中  */
asm {
    /*  将低8位写入行比较寄存器  */
    mov bx,hi
    mov dx,CRT_CTL
    mov al,18h
    out dx,al
    inc dx
    mov al,bl
    out dx,al
    /*  将位8写入溢出寄存器的位4  */
    dec dx
    mov al,7
    out dx,al
    inc dx
    in al,dx
    mov ah,bh
    mov cl,4
    shl ah,cl
    or ah,11101111b
    and ah,al
    dec dx
    mov al,7
    out dx,al
    inc dx
    mov al,ah
    out dx,al
    /*  将位9写入最大扫描线寄存器的位6  */
    dec dx
    mov al,9
    out dx,al
    inc dx
    in al,dx
    mov ah,bh
    mov cl,5
    shl ah,cl
    or ah,10111111b
    and ah,al
    dec dx
    mov al,9
    out dx,al
    inc dx
    mov al,ah
    out dx,al
    }
}
```

```
/*  在正常模式设置的基础上设置漫游模式  */
void ROAM::setmode()
{
setScreenWide(VWIDE);
cutScreen(activHigh-1);
setbxy(0,0);
}

/*  检验、调整、设置有关的屏幕参数  */
void ROAM::checkpar(void)
{
long l;
int maxy;
activHigh=orgHigh-fixedHigh;
if( activHigh<=0 )
    activHigh=0;
if( activHigh>=orgHigh )  //活动屏幕太高
    activHigh=orgHigh-1;
if( activHigh>0 && activHigh<orgHigh )
    fixedHigh=orgHigh-activHigh;
else
    fixedHigh=0;

if(VWIDE>255*minSetWide)  //虚屏宽度超出了偏移量的范围
    VWIDE=255*minSetWide
VWIDE=(VWIDE/minSetWide)*minSetWide;
  //调整虚屏宽度，使其为偏移量的单位象素点数的整数倍
if(VWIDE<orgWide)  //虚屏宽度不能小于实际屏幕的宽度
    VWIDE=orgWide;
maxMoveX=VWIDE-orgWide;  //设置x方向的最大移动范围

l = 0xffffL*(long)minMoveWide;
l /= VWIDE;
maxy = l+activHigh;  //虚屏最大能够具有的高度
if(VHIGH>maxy)  //虚屏高度太大
    VHIGH=maxy;
if(VHIGH<orgHigh)  //虚屏高度不能小于实际屏幕的高度
    VHIGH=orgHigh;

maxMoveY=VHIGH-activHigh-1;  //设置y方向移动范围的高限
minMoveY=orgHigh-activHigh-1;  //设置y方向移动范围的低限
if(activHigh==0||fixedHigh==0)
    minMoveY=0;
}

static unsigned char _roam_key_value[2][4]={{72,80,75,77},{200,208,203,205}};
  //4个方向键按下和放开时的键盘扫描码
void interrupt (*oldkbint0x09)(void);
```

```
/*  键盘自动漫游功能  */
void interrupt ROAM::kbint0x09_roam()
{
static unsigned mesp,mess,invalue,i,k,kb,f224=0,dx,dy;
disable();
mesp=_SP; mess=_SS;
enable();
invalue=inportb(0x60);
if(invalue==224)
    {
    k=0;
    f224=1;
    }
else if(f224==1)
    {
    f224=0;
    k=1;
    dx=dy=0;
    for(i=0;i<4;i++)
        if(invalue==_roam_key_value[1][i])
            k=0;
    for(i=0;i<4;i++)
        if(invalue==_roam_key_value[0][i])  //检查某个方向键是否按下
            {
            k=0;
            if(i==0) dy=-dxy;
            else if(i==1) dy=dxy;
            else if(i==2) dx=-dxy;
            else if(i==3) dx=dxy;
            roam(dx,dy);
            break;
            }
    }
else
    k=1;
disable();
_SP=mesp; _SS=mess;
enable();
if(k)
    (*oldkbint0x09)();  //调用原键盘中断功能
else
    {
    kb=inportb(0x61); outportb(0x61,0x80);
    outportb(0x61,kb); outportb(0x20,0x20);
    }
}

/*  安装键盘自动漫游功能，设置每次按键的移动量  */
```

```
void ROAM::setup_key_roam(int d)
{
dxy=d;
if(KEY_OK==0)
    {
    oldkbint0x09=(void interrupt(*)())getvect(0x09);
    disable();
    setvect(0x09,(void interrupt(*)(...))kbint0x09_roam);
    enable();
    KEY_OK=1;  //设置已安装的标志
    }
}

/*  删除键盘自动漫游功能。如果程序安装了键盘自动漫游功能，那么在程序退出前就必须删除它，否则程序退出后就将不能进行正常的键盘输入  */
void ROAM::del_key_roam()
{
if(KEY_OK)
    {
    disable();
    setvect(0x09,(void interrupt(*)(...))oldkbint0x09);
    enable();
    KEY_OK=0;
    }
}
```

14.3.2　16色模式下的漫游程序

每个漫游模式类中都只需定义两个函数，一个是构造函数，一个是模式设置函数。

构造函数用于接收、检验和调整应用程序所指定的虚屏宽度、虚屏高度及屏幕分割方式，并设置其它的有关参数，这包括：实际屏幕的宽度和高度，屏幕横向移动的最小单位、设置屏幕宽度的最小单位，虚屏中每条扫描线所占字节数，虚屏所占显示存储器页数，原图形模式的模式号。

每个漫游模式类的模式设置函数都是一样的，其只包含三行语句，首先设置正常的图形模式，然后设置漫游模式，最后将整个虚屏清零。

系统程序14-3　ROAM16.CPP

```
/*************************************************************************
功能：构造函数。设置参数
输入参数：  vwi＝虚屏的宽度
            vhi＝虚屏的高度
            fixedhi＝固定屏幕的高度。该值为0，即不分割屏幕。该参数可以缺省，缺省值为0
*************************************************************************/
_640_480_16_roam::_640_480_16_roam(int vwi,int vhi,int fixedhi)
{
long l;
orgWide=640;  //实际屏幕的宽度
```

```
orgHigh=480;  //实际屏幕的高度
minMoveWide=8;  //屏幕横向移动的最小单位
minSetWide=16;  //设置屏幕宽度的最小单位
/*  以下12行，将虚屏的大小调整到256K以内  */
l=(long)(vwi)*(long)vhi;
if(l>0x80000L)
    {
    l=480L*(long)vwi;
    if(l>0x80000L)
        {
        vwi=0x80000L/480L;
        vhi=480;
        }
    else
        vhi=0x80000L/vwi;
    }
VWIDE=vwi;  //虚屏宽度
VHIGH=vhi;  //虚屏高度
fixedHigh=fixedhi;  //固定屏幕的高度
checkpar();  //检验、调整参数
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE/8;  //虚屏中一条扫描线所占字节数
PAGEN=1;  //虚屏所占显示存储器页数
VESAmodeNo=0x0012;  //原图形模式的模式号
}

/*  模式设置  */
void _640_480_16_roam::setmode()
{
VGABASE::setmode();  //正常模式设置
ROAM::setmode();  //漫游模式设置
cls0();  //虚屏清零
}

_800_600_16_roam::_800_600_16_roam(int vwi,int vhi,int fixedhi)
{
orgWide=800;
orgHigh=600;
minMoveWide=8;
minSetWide=16;
/*  因16色模式下的图形操作函数不能处理行外分页情况，因此下面4行将虚屏宽度限制在1024或
2048，以避免行外分页  */
if(vwi<1536)
    vwi=1024;
else
    vwi=2048;
VWIDE=vwi;
VHIGH=vhi;
```

```
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE/8;
PAGEN=4;
VESAmodeNo=0x0102;
}

void _800_600_16_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}

_1024_768_16_roam::_1024_768_16_roam(int vwi,int vhi,int fixedhi)
{
orgWide=1024;
orgHigh=768;
minMoveWide=8;
minSetWide=8;
vwi=1024;  //虚屏宽度固定为1024。其不可能达到2048
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE/8;
PAGEN=4;
VESAmodeNo=0x0104;
}

void _1024_768_16_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}
```

14.3.3　256色模式下的漫游程序

系统程序14-4　ROAM256.CPP

```
_640_400_256_roam::_640_400_256_roam(int vwi,int vhi,int fixedhi)
{
orgWide=640;
```

```
orgHigh=400;
minMoveWide=8;
minSetWide=16;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE;
PAGEN=16;
VESAmodeNo=0x0100;
}

void _640_400_256_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}

_640_480_256_roam::_640_480_256_roam(int vwi,int vhi,int fixedhi)
{
orgWide=640;
orgHigh=480;
minMoveWide=8;
minSetWide=16;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE;
PAGEN=16;
VESAmodeNo=0x0101;
}

void _640_480_256_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}

_800_600_256_roam::_800_600_256_roam(int vwi,int vhi,int fixedhi)
{
orgWide=800;
orgHigh=600;
```

```
minMoveWide=8;
minSetWide=16;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE;
PAGEN=16;
VESAmodeNo=0x0103;
}

void _800_600_256_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}

_1024_768_256_roam::_1024_768_256_roam(int vwi,int vhi,int fixedhi)
{
orgWide=1024;
orgHigh=768;
minMoveWide=8;
minSetWide=8;
//vwi=1024;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE;
PAGEN=16;
VESAmodeNo=0x0105;
}

void _1024_768_256_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}
```

14.3.4　高彩色模式下的漫游程序

系统程序14-5　ROAMHIGH.CPP

```
--------------------------------------------------------------------------
_320_200_32k_roam::_320_200_32k_roam(int vwi,int vhi,int fixedhi)
{
orgWide=320;
orgHigh=200;
minMoveWide=2;
minSetWide=4;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE*2;
PAGEN=16;
VESAmodeNo=0x010d;
}

void _320_200_32k_roam::setmode()
{
VGABASE::setmode();
setScreenWide(WIDE);
cutScreen(0);
setbxy(0,0);
cls0();
}

_512_480_32k_roam::_512_480_32k_roam(int vwi,int vhi,int fixedhi)
{
orgWide=512;
orgHigh=480;
minMoveWide=2;
minSetWide=4;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE*2;
PAGEN=16;
VESAmodeNo=0x0170;
}

void _512_480_32k_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
```

```
}

_320_200_64k_roam::_320_200_64k_roam(int vwi,int vhi,int fixedhi)
{
orgWide=320;
orgHigh=200;
minMoveWide=2;
minSetWide=4;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE*2;
PAGEN=16;
VESAmodeNo=0x010e;
}

void _320_200_64k_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}

_512_480_64k_roam::_512_480_64k_roam(int vwi,int vhi,int fixedhi)
{
orgWide=512;
orgHigh=480;
minMoveWide=2;
minSetWide=4;
VWIDE=vwi;
VHIGH=vhi;
fixedHigh=fixedhi;
checkpar();
WIDE=VWIDE;
HIGH=VHIGH;
SCANLENG=VWIDE*2;
PAGEN=16;
VESAmodeNo=0x0171;
}

void _512_480_64k_roam::setmode()
{
VGABASE::setmode();
ROAM::setmode();
cls0();
}
```

14.4 程序使用方式

各个漫游模式类的使用方式与各图形模式类的使用方式是完全相同的，进行字符显示、鼠标驱动及屏幕硬拷贝的方式也是完全相同的，只是在定义一个具体的漫游模式类对象时需要指定虚屏的宽度、高度及屏幕分割参数。

如果要在程序中使用键盘自动漫游功能，只需在定义了一个漫游模式对象之后运行ROAM::setup_key_roam()函数，在程序退出前运行ROAM::del_key_mouse()函数，但要注意在使用键盘自动漫游功能时就不能使用键盘模拟鼠标，因为它们使用了相同的方向控制键，并使用了相同的中断功能。

如果要使用鼠标自动漫游功能，则需在鼠标光标显示类中增加两行语句，一是在构造函数中增加一行：activMask=MouseMove; 二是在事件处理函数active(Event ev)中增加一行：ROAM::autoroam(ev.mouse.x,ev.mouse.y,wide,high)。可以从一个已有的光标显示类中派生出一个类，然后在其中增加这两行语句。下面给出两个分别使用键盘和鼠标作自动漫游的示例程序。

示例程序14-1 使用键盘自动漫游，并作屏幕分割

```
void main()
{
Event ev;
_800_600_256_roam roam(1024,1024,100);
  //定义一个1024×1024的虚屏，并分割屏幕，固定屏幕的高度为100
roam.init();
roam.setup_key_roam();  //安装键盘自动漫游功能
roam.setcolor(7);
roam.bar(0, 0, roam.orgWide-1, roam.fixedHigh-1);  //在固定屏幕中画一个实矩形
roam.setcolor(4);
/*  在活动屏幕中画两条交叉的斜直线  */
roam.line(0, roam.fixedHigh, roam.WIDE-1, roam.HIGH-1);
roam.line(roam.WIDE-1, roam.fixedHigh, 0, roam.HIGH-1);
ev.getevent( evKey );  //等待一个按键事件，在期间用户即可用4个方向键移动屏幕
  //当按下一个其它键时，程序即退出
roam.del_key_roam();  //删除键盘自动漫游功能
roam.close();  //关闭漫游模式
}
```

示例程序14-2 使用鼠标自动漫游

```
/*  直接从已有的一个鼠标光标显示类中派生出一个支持鼠标自动漫游的光标显示类  */
class MMap16_roam : public MouseMap16
    {
public:
    MMap16_roam(VGABASE *v) : MouseMap16(v)
        { activMask=MouseMove; }
protected:
    void active(Event evt)
        { ROAM::autoroam(evt.mouse.x,evt.mouse.y,wide,high); }
```

```
    } ;

void main()
{
Event ev;
_640_480_256_roam roam(1024,2024);
  //定义一个1024×1024的虚屏，实际得到的虚屏大小将为1024×991
roam.init();
MOUSE::init();
MMap16_roam mmap(&roam);
roam.setcolor(4);
roam.line(0, 0, roam.WIDE-1, roam.HIGH-1);
roam.line(roam.WIDE-1, 0, 0, roam.HIGH-1);
MOUSE::show();
ev.getevent( MouseDouble );
  //等待发生鼠标左按钮双击事件，这期间用户即可用鼠标控制屏幕移动
MOUSE::hide();
roam.close();
}
```

第15章　程 序 说 明

本书附带一张软盘，它包含了本书所实现的VGA图形支持系统的全部源程序，及一个直接供软件开发用的类库，它还包含了一些与本书有关的实用程序及它们的源程序。这一章对该软盘的内容及其所包含的各类程序的使用方法进行说明。

15.1　软 盘 内 容

15.1.1　安装

软盘中的程序需安装到硬盘上才能使用。软盘中只包含有一个文件VGA.EXE，这是一个可自解的压缩文件，本书所提供的全部程序都包含在其中，直接运行该文件即可实现解压和安装，执行步骤如下：

①将软盘插入A或B驱动器，这里假设为B驱动器；

②假设要将程序安装到逻辑硬盘D，在D盘的根目录下键入命令：B:VGA[CR]，以运行VGA.EXE；

③对运行过程中所出现的任何提问都键入“Y”，并回车。运行完后，全部程序就都安装到了D盘上。

安装完后，硬盘上将出现一个如图15-1所示的目录树，所提供的各个程序被分类存放在一定的子目录中。你可以重新组织整个程序的存放方式，但这时就需对各个源程序作两方面的修改，一是，如果改变了\VGA目录中各头文件的存放位置，就需修改各源程序中所指定的头文件目录；二是，如果改变了\VGA\FONT目录中各字库的存放位置，就需修改有字符显示的那些实用程序中所指定的字库文件的目录。

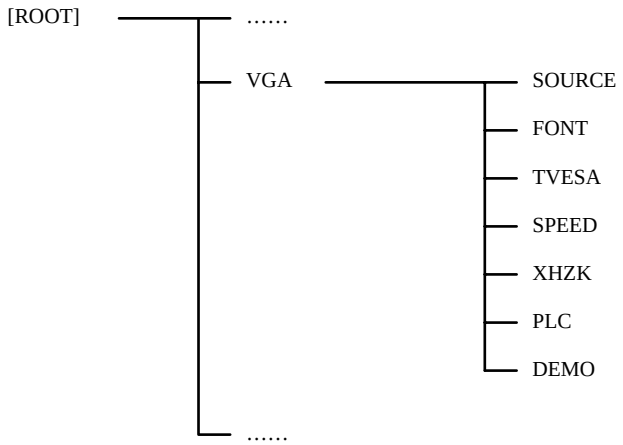

图15-1　目录结构

15.1.2　文件列表

下面按存放的目录分类列出所提供的各个程序文件。

1. \VGA 目录

该目录下存放着1个库文件和13个头文件，它们构成了本书所实现的VGA基础图形系统。在应用软件开发中所需使用的主要就是这14个文件。

VGAL.LIB	大模式的基础图形类库
XMS.H	扩展内存操作类说明
VGABASE.H	基本图形类说明
FILLBASE.H	基本填充类说明
VGA16.H	16色模式类说明
VGA256.H	256色模式类说明
VGAHIGH.H	高彩色模式类说明
VGA16M.H	16M色模式类说明
VGAFONT.H	字符显示类说明
VGAPRT.H	图形打印类说明
VGAMOUSE.H	鼠标光标显示类，鼠标操作类说明
EVTCODE.H	事件代码定义
VGAROAM.H	屏幕漫游类说明
VGA.H	其包含了上面12个头文件。在应用程序中只需包含这一个头文件

在应用程序的工程文件中加入VGAL.LIB，在源程序中包含进头文件“VGA.H”，即可在应用程序中使用本书所提供的所有基础图形程序。

2. \VGA\COURCE 目录

该目录下存放着VGA基础图形系统的全部源文件，共22个，编译这些源文件，然后用Borland的库管理程序TLIB.EXE，将所生成的目标模块全部放到一个库文件中，即可得到基础图形类库VGAL.LIB，该目录还包含一个批处理文件和一个工程文件。

XMS.CPP	扩展内存操作程序
VGABASE.CPP	VGA基本操作程序
VGADRAW.CPP	绘图程序
FILLBASE.CPP	基础填充程序
VGAFILL.CPP	填充程序
16.CPP	16色模式下的图形操作程序
256.CPP	256色模式下的图形操作程序
HIGH.CPP	高彩色模式下的图形操作程序
16M.CPP	真彩色模式下的图形操作程序
FONT.CPP	字符显示基本程序及点阵字符显示程序
FONTSL.CPP	矢量字符显示程序
PRT.CPP	图形打印程序
MOUSE.CPP	鼠标操作程序及鼠标光标显示基础程序
MOUSE16.CPP	16色模式下的鼠标光标显示程序
MOUSE256.CPP	256色模式下的鼠标光标显示程序
MOUSEHI.CPP	高彩色模式下的鼠标光标显示程序
MOUSE16M.CPP	真彩色模式下的鼠标光标显示程序
KEYMOUSE.CPP	键盘模拟鼠标程序

ROAM.CPP　　屏幕漫游基础程序

ROAM16.CPP　16色模式的漫游程序

ROAM256.CPP　256色模式的漫游程序

ROAMHIGH.CPP　高彩色模式的漫游程序

ALL.PRJ　包含编译设置的工程文件

CRLIB.BAT　建立类库的批处理文件

ALL.PRJ是一个borland C++ 3.1格式的工程文件，它并不对应于一个执行文件，它的主要作用是保存类库中目标模块的编译设置。CRLIB.BAT用于将所编译出的各目标模块生成为类库。

3. \VGA\FONT 目录

该目录用于存放各种字库，本书只提供了三个ASCII字库。如果能获得本书的字符显示程序所支持的其它字库，应将它们拷到该目录下。

ASCK16　16点阵ASCII字库

ASCK24　24点阵ASCII字库

ASKSL　ASCII轮廓矢量字库

4. \VGA\TVESA 目录

TVESA.EXE　VESA图形模式检测程序

TVESA.CPP　源程序

5. \VGA\SPEED 目录

VS.EXE　各种模式下图形显示速度测试程序

VS.CPP　源程序

VS.PRJ　工程文件

6. \VGA\XHZK 目录

XHZK.EXE　小汉字库构造程序

XHZK.CPP　源程序

7. \VGA\PLC 目录

PLC.EXE　大字打印程序

PLC.CPP　源程序

8. \VGA\DEMO 目录

VGADEMO.EXE　演示程序

VGADEMO.CPP　源程序。主控程序

DMODE.CPP　　源程序。演示图形模式

DFONT.CPP　　源程序。演示字符显示

DPRT.CPP　　源程序。演示图形打印

DMOUSE.CPP　源程序。演示鼠标

DROAM.CPP　　源程序。演示屏幕漫游

VGADEMO.PRJ　工程文件

15.2 类 库 说 明

基础图形类库VGAL.LIB中所包含的各个类都是在本书前面的各章节中实现的，在那些章节中已经对各个类进行了不同程度的说明，但所作的说明主要是从程序设计和实现的角度进行的，而且是分散的，这里将从使用的角度集中对各个类作出系统的说明，以便于读者使用该类库。

对一个已有的类，存在着两种使用方式，一是调用，二是继承，这里的说明主要针对于第一种使用方式。如果一个应用程序仅仅只是调用类库中的类，这时类库中的那些抽象类及非抽象类中受保护的成员对应用程序来说就是不可见的，应用程序也就没有必要了解类库中的这些成分，因此下面只说明类库中的那些实例类，并且只说明它们的公共(public)成员。

当要调用一个类时，我们所关心的只是这个类有那些成员可以调用，而并不关心它的每个成员到底是由它本身定义的还是从它的父类那里继承来的，因此，这里将对每个实例类的所有可用成员进行说明，而不仅仅是说明它本身所定义的那些成员。

类库中的那些具有相同父类的类，它们的成员大部分都是相同的，对这些类将统一进行说明，那些具有相同父类的类称为一个类族，下面将按类族对类库中的各个实例类进行说明，对每个类族将给出它所包含的实例类的数量、每个实例类的类名、它的各个基类以及有关的说明，然后列出它的所有可用数据成员及成员函数。

15.2.1 扩展内存类

数量：1

类名：XMS

基类：无

说明：用于扩展内存操作。它的一个对象对应于一块已分配的扩展内存，该内存块随着对象的定义而分配，随着对象的删除而释放。对一个扩展内存块的所有操作都需要通过该类的一个对象来进行，那些对整个扩展内存的操作函数，则被说明为静态的，它们不需要通过具体的对象来调用。

数据成员：

```
static int OK
```

整个扩展内存是否可用的标志，1表示可用，0表示不可用。

成员函数：

```
XMS(int size)
```

分配一个以k字节计、大小为size的扩展内存块。这是构造函数，在定义一个对象被自动调用。

```
~XMS()
```

释放一个对象所对应的扩展内存块。这是析构函数，在删除对象时被自动调用。

```
static int init(void)
```

初始化扩展内存操作程序。在进行任何扩展内存操作之前，必须首先运行此函数。

static unsigned freesize(void)

获取以k字节计的自由扩展内存的大小。

static unsigned largestblock(void)

获取以K字节计的最大自由扩展内存块的大小。

int realloc(int size)

按尺寸size重新分配扩展内存块的大小。

int put(void *dp,void *sp,long leng)

将常规内存中的一块数据放到扩展内存块中。leng为数据块的长度，sp为数据块在常规内存中的地址，dp为数据块放到扩展内存块中的位置。

int get(void *dp,void *sp,long leng)

将扩展内存块中的一段数据取到常规内存中。dp为常规内存中的地址，sp为数据在扩展内存块中的位置。

15.2.2　图形模式类

数量：18

类名：_640_480_16　　640×480×16色图形模式类
　　_800_600_16　　800×600×16色图形模式类
　　_1024_768_16　　1024×768×16色图形模式类
　　_320_200_256　　320×200×256色图形模式类
　　_640_400_256　　640×400×256色图形模式类
　　_640_480_256　　640×480×256色图形模式类
　　_800_600_256　　800×600×256色图形模式类
　　_1024_768_256　　1024×768×256色图形模式类
　　_320_200_32k　　320×200×32K高彩色图形模式类
　　_512_480_32k　　512×580×32K高彩色图形模式类
　　_640_480_32k　　640×480×32K高彩色图形模式类
　　_800_600_32k　　800×600×32K高彩色图形模式类
　　_320_200_64k　　320×200×64K高彩色图形模式类
　　_512_480_64k　　512×580×64K高彩色图形模式类
　　_640_480_64k　　640×480×64K高彩色图形模式类
　　_800_600_64k　　800×600×64K高彩色图形模式类
　　_320_200_16M　　320×200×16M真彩色图形模式类
　　_640_480_16M　　640×480×16M真彩色图形模式类

基类：VGABASE、VGAindirect、VGA16、VGA256、VGAHIGH、VGA32K、VGA64K、VGA16M

说明：用于图形显示。每一个类对应于VGA的一种图形模式，定义某个类的一个对象，即可通过该对象调用相应一种图形模式下的所有图形显示功能。

数据成员：

int OK

图形模式初始化是否成功的标志。1表示初始化成功，可以进行图形显示；0表示初始化失败或没有进行初始化，不能进行图形显示。

COLOR CUR_COLOR

当前图形显示的颜色。COLOR是一联合，其可用于存放各种色彩模式下的颜色值。

int WIDE

当前图形模式下的屏幕水平象素数。

int HIGH

当前图形模式下的屏幕垂直象素数。

int VESAmodeNo

当前图形模式的VESA模式号或标准VGA模式号。

成员函数：

构造函数(void)

设置图形模式的有关参数。

virtual int init(void)

图形模式初始化。在进行图形显示之前必须首先运行此函数，其返回初始化是否成功的标志，1表示成功，0表示失败。

void close(void)

关闭图形模式，回到缺省的字符显示模式。

void display_off(void)

关闭屏幕刷新，以使CPU对显示存储器的访问达到最快。

void display_on(void)

打开屏幕刷新，恢复正常的屏幕显示。

virtual void putpixel(int x,int y)

以当前颜色写一个象素点。

virtual void putpixel(int x,int y,COLOR color)

以颜色color写一个象素点。

virtual union COLOR getpixel(int x,int y)

读一个象素点的颜色值。以COLOR联合的格式返回所读出的颜色值。

virtual void scanline(int x1,int x2,int y)

以当前颜色画一条水平直线。

virtual void scanline(int x1,int x2,int y,COLOR color)

以颜色color画一条水平直线。

virtual void cls0(void)

置零清屏。以页为单位，将当前图形模式所占用的显示存储器全部清零。

virtual void cls(void)

置色清屏。用当前颜色写满整个屏幕。

virtual void cls(COLOR color)

用颜色color写满整个屏幕。

virtual void getscanline(int x1,int y,int n,void *buf)

读扫描线。将开始于(x1,y)点，长度为n的扫描线读到内存缓冲区buf中。

virtual void putscanline(int x1,int y,int n,void *buf)

写扫描线。将内存缓冲区buf中的数据写到开始于(x1,y)点，长度为n的扫描线上。

virtual int scanlinesize(int x1,int x2)

取保存一条扫描线所需缓冲区的大小，扫描线两个端点的横坐标为x1和x2。

void line(int x1,int y1,int x2,int y2)

以当前颜色画一条直线，直线的两个端点为(x1,y1)和(x2,y2)。

void line(int x1,int y1,int x2,int y2,COLOR color)

以颜色color画一条直线。

void lineto(int x,int y)

以当前颜色在画线开始点至(x,y)点间画一条直线。

void lineto(int x,int y,COLOR color)

以颜色color在画线开始点至(x,y)点间画一条直线。

void moveto(int x,int y)

设置画线开始点。所有的画直线函数都会自动将画线开始点设为其所画直线的末点。

void rectangle(int x1,int y1,int x2,int y2)

以当前颜色画一个矩形。矩形左上角的坐标为(x1,y1)，右下角的坐标为(x2,y2)。

void rectangle(int x1,int y1,int x2,int y2,COLOR color)

以颜色color画一个矩形。

void poly(int n,int *border)

以当前颜色画一个多边形。

void poly(int n,int *border,COLOR color)

以颜色color画一个多边形。

void poly(int *border)

以当前颜色画多个多边形。

void poly(int *border,COLOR color)

以颜色color画多个多边形。

void circle(int x0,int y0,int r)

以当前颜色画一个圆心为(x0,y0)，半径为r的圆。

void circle(int x0,int y0,int r,COLOR color)

以颜色color画一个圆。

void sector(int x0,int y0,int r,int stangle,int endangle)

以当前颜色画一个扇形。扇形的起始角度为stangle，终止角度为endangle，将按逆时针方向从起始角度画到终止角度。

void sector(int x0,int y0,int r,int stangle,int endangle,COLOR color)

以颜色color画一个扇形。

void ellipse(int x0,int y0,long r1,long r2)

以当前颜色画一个椭圆，椭圆心为(x0,y0)，横半轴长为r1，纵半轴长为r2。

void ellipse(int x0,int y0,long r1,long r2,COLOR color)

以颜色color画一个椭圆。

void setfillstyle(int fst)

设置当前填充模式。填充模式代码fst=0～11。

int getfillstyle(void)

获取当前的填充模式

void setfillpattern(unsigned char *s)

设置用户自定义填充图案

void fillarea(int x0,int y0,COLOR bordercolor)

以当前颜色和当前填充模式填充一个任意的封闭区域。(x0,y0)为区域内部的任意一点，bordercolor为区域边界的颜色。

void fillarea(int x0,int y0,COLOR bordercolor,COLOR color)

以颜色color和当前填充模式填充一个任意的封闭区域。

void bar(int x1,int y1,int x2,int y2)

以当前颜色和当前填充模式画一个实矩形。

void bar(int x1,int y1,int x2,int y2,COLOR color)

以颜色color和当前填充模式画一个实矩形。

void polyfill(int n,int *border)

以当前颜色和当前填充模式填充一个多边形。

void polyfill(int n,int *border,COLOR color)

以颜色color和当前填充模式填充一个多边形。

void polyfill(int *border)

以当前颜色和当前填充模式同时填充多个多边形。

void polyfill(int *border,COLOR color)

以颜色color和当前填充模式同时填充多个多边形。

void circlefill(int x0,int y0,int r)

以当前颜色和当前填充模式画一个实圆。

void circlefill(int x0,int y0,int r,COLOR color)

以颜色color和当前填充模式画一个实圆。

void sectorfill(int x0,int y0,int r,int stangle,int endangle)

以当前颜色和当前填充模式画一个实扇形。

void sectorfill(int x0,int y0,int r,int stangle,int endangle,COLOR color)

以颜色color和当前填充模式画一个实扇形。

void ellipsefill(int x0,int y0,long r1,long r2)

以当前颜色和当前填充模式画一个实椭圆。

void ellipsefill(int x0,int y0,long r1,long r2,COLOR color)

以颜色color和当前填充模式画一个实椭圆。

IMAGE *getimage(int x1,int y1,int x2,int y2,int where=inMEM)

读出一屏幕矩形块，将数据存于常规内存(inMEM)、扩展内存(inXMS)或硬盘(inHD)。其返回一个IMAGE结构的指针，应用程序应保存该指针以用于恢复矩形块。该函数本身完成存储区域的分配，当所读出的数据不再使用时，应用程序应删除其所返回的IMAGE结构，以释放存储空间。

void putimage(int x1,int y1,IMAGE *img)

在一个新的位置恢复所读出的矩形块，新位置左上角的坐标为(x1,y1)，img为所读出矩形块的标识。

void putimage(IMAGE *img)

在原处恢复所读出的矩形块。

void getimageMEM(int x1,int y1,int x2,int y2,void *buf)

读出一屏幕矩形块，数据存于常规内存中的一个缓冲区buf中，由应用程序负责分配和释放buf缓冲区。

void putimageMEM(int x1,int y1,int xn,int yn,void *buf)

将用getimageMEM()函数所读出的矩形块恢复到屏幕上，这时应指定矩形块左上角的位置(x1,y1)，及矩形块的宽度xn、高度yn，其必须与读出时一致。

long imagesize(int x1,int y1,int x2,int y2)

获取保存一个屏幕矩形块所需缓冲区的大小。

上面的各个函数在所有的18个图形模式类中都是可用的，下面一些函数则只在部分图形模式类中可用。

virtual void setcolor(unchar color);

设置当前图形显示颜色。在间接色彩模式下，所输入的一个字节为颜色索引值，在高彩色模式下则为灰度值，在真彩色模式下该函数不能使用。

virtual COLOR setcolorto(unchar color)

进行颜色格式转换，将一个字节的颜色值转换为COLOR格式的颜色值。其在真彩色模式下不能使用。

virtual void setcolor(unchar r,unchar g,unchar b)

设置当前图形显示颜色，所输入的三个字节依次为红、绿、蓝三基色的亮度值，其在间接色彩模式下无效。

virtual COLOR setcolorto(unchar r,unchar g,unchar b)

转换颜色格式。只在间接色彩模式下无效。

void setdac(unchar idx,unchar r,unchar g,unchar b)

设置DAC寄存器。idx为颜色索引值，其余三个字节依次为红、绿、蓝三色的亮度值，亮度值的取值范围为0～255。该函数只在间接色彩模式下可用。

void getdac(unchar idx,unchar &r,unchar &g,unchar &b)

读DAC寄存器。该函数只在间接色彩模式下可用。

void setpalette(unchar idx16,unchar idx256)

设置调色板寄存器。该函数只在16色模式下可用。

unchar getpalette(unchar idx16)

读调色板寄存器。该函数只在16色模式下可用。

以下为真彩色模式下的15个颜色变换函数，在其它色彩模式下它们都是不可用的。

void en_putpixel(int x,int y,unchar enl,unchar redu)

改变一个象素点的亮度，保持色调不变。

void en_scanline(int x1,int x2,int y,unchar enl,unchar redu)

改变一条扫描线的亮度，保持色调不变。

void en_bar(int x1,int y1,int x2,int y2,unchar enl,unchar redu)

改变一个矩形块的亮度，标尺色调不变。

void re_putpixel(int x,int y)

将当前颜色叠加写到一个象素点上。

void re_putpixel(int x,int y,COLOR color)

将颜色color叠加写到一个象素点上。

void re_scanline(int x1,int x2,int y)

将当前颜色叠加写到一条扫描线上。

void re_scanline(int x1,int x2,int y,COLOR color)

将颜色color叠加写到一条扫描线上。

void re_bar(int x1,int y1,int x2,int y2)

将当前颜色叠加写到一个矩形块上。

void re_bar(int x1,int y1,int x2,int y2,COLOR color)

将颜色color叠加写到一个矩形块上。

void de_putpixel(int x,int y)

去除叠加在一个象素点上的当前颜色。

void de_putpixel(int x,int y,COLOR color)

去除叠加在一个象素点上的颜色color。

void de_scanline(int x1,int x2,int y)

去除叠加在一条扫描线上的当前颜色。

void de_scanline(int x1,int x2,int y,COLOR color)

去除叠加在一条扫描线上的颜色color。

void de_bar(int x1,int y1,int x2,int y2)

去除叠加在一个矩形块上的当前颜色。

void de_bar(int x1,int y1,int x2,int y2,COLOR color)

去除叠加在一个矩形块上的颜色color。

15.2.3 字符显示类

数量：21

类名：ASC16 16点阵ASCII字库字符显示类

ASC24 24点阵ASCII字库字符显示类

ASCBC Borland ASCII矢量字库字符显示类

ASCSL ASCII轮廓矢量字库字符显示类

HZ16 16点阵汉字库字符显示类

HZ24 24点阵汉字库字符显示类

TX24 24点阵图形汉字库字符显示类

HZ32 32点阵汉字库字符显示类

TX32 32点阵图形汉字库字符显示类

HZ40 40点阵汉字库字符显示类

TX40 40点阵图形汉字库字符显示类

HZUC UCDOS矢量汉字库字符显示类

HZUCK UCDOS矢量楷体汉字库字符显示类

TXUC UCDOS矢量图形汉字库字符显示类

HZ16_S　16点阵小汉字库字符显示类
HZ24_S　24点阵小汉字库字符显示类
HZ32_S　32点阵小汉字库字符显示类
HZ40_S　40点阵小汉字库字符显示类
HZUC_S　矢量小汉字库字符显示类
HZUCK_S　矢量楷体小汉字库字符显示类
TXUC_S　矢量图形小字库字符显示类

基类：FONT、FONT_ASC

说明：用于字符显示。每个字符显示类对应于一种类型的字库，某个类的一个对象则对应于相应类型的一个具体字库，定义一个对象，即可通过该对象来使用某个字库进行字符显示。一个字符显示类对象与一个图形模式类对象相连接来实现各种图形模式下的字符显示，如果所连接的是一个打印图形缓存类对象，则可实现字符打印。

数据成员：

int OK

一个对象所对应的字库是否装载成功的标志，1表示成功，该对象可以进行字符输出；0表示失败，该对象不能进行字符输出。

int where

字库装载的区域，其可能的取值为inMEM、inXMS、inHD或inNONE，如果为inNONE，则表示字库装载失败。

long leng

以字节计的字库的长度。

成员函数：

构造函数(char *filename,int whe,VGABASE *v)

装载字库并与一个图形模式类对象相连接。filename为字库文件名，whe为指定的装载区域，当所指定的区域容量不够时，将自动转到使用一个更大的存储区域，实际的装载区域可从数据成员where中获得，v为一个图形模式类对象的指针，其也可为图象缓存类对象或漫游模式类对象的指针。该函数在定义一个对象时自动运行。

析构函数

释放字库所占的空间。这是析构函数，在删除对象时自动运行。

void setvga(VGABASE *v)

重新设置于字符显示相连接的图形模式类对象。

void join(FONT *ftp1,FONT *ftp2=0)

将其它字符集的字库连接进来，以能在一个字符串中输出不同字符集中的字符。

void cut(FONT *ftp);

切断与其它某个字符集字库之间的连接。

void setsize(int hi,int wi,int xxi=0)

设置所输出字符的高度hi、宽度wi及字符串中的字符间隔xxi，其中宽度和高度仅对矢量字库有效。其将同时对连接进来的其它字库进行设置。

void setsizeOnly(int hi,int wi,int xxi=0)

设置字符宽度、高度和间隔，其不对连接进来的字库进行设置。

int gettexthigh()

获得字符输出的高度。

virtual int gettextwide(char *str)

获得一个字符串的宽度。

virtual void outtextxy(int x,int y,char *string)

在指定位置输出一个字符串，所用颜色为当前图形显示颜色。

void putxy(int x,int y)

设置字符显示的当前位置。

void getxy(int &x,int &y)

获得字符显示的当前位置。

void outtext(char *string)

在当前位置用当前颜色输出一个字符串。

void printf(char *fmt, ...)

在当前位置用当前颜色格式化输出一字符串。

void sethollow(int k)

设置轮廓矢量字符的空心显示。1表示空心。

virtual int getcharwide(unsigned char c)

获取一个字符的宽度。

virtual int outchar(int x,int y,unsigned char *cp)

在指定位置输出一个字符。

15.2.4 图形打印缓存类

数量：1

类名：prt_buf

基类：VGABASE

说明：用于建立图形打印缓存区，并在缓存区中进行图形操作。该类的一个对象对应于一个图象缓存区，缓存区可建在常规内存中、扩展内存中或硬盘中。该类继承于基础图形类VGABASE，因此它具有图形模式类所具有的所有图形操作功能及绘图功能，并且能与字符显示类相连接。当要进行图形打印时，首先应用该类建立一个图象缓存区，并在缓存区上完成所有的图形操作，然后用图象打印类将缓存区中的图象输出到打印机上。

数据成员：

该类的数据成员与各图形模式类的数据成员完全相同。

成员函数：

除了那些为部分图形模式类所拥有的成员函数之外，该类拥有图形模式类的其它所有成员函数。此外，该类增加了如下两个成员函数。

prt_buf(int wi,int hi,int whe=inMEM)

在whe所指定的区域建立一个宽度为wi、高度为hi的图象缓存区，当所指定的区域容量不够时，自动转到在容量更大的区域建立缓存区。这是构造函数，其在定义一个对象时被自动调用。

~prt_buf()

释放图象缓存区。其在删除对象时被自动调用。

15.2.5　图象打印类

数量：8

类名：EPSON_360　EPSON系列打印机360×360dpi图象打印类
EPSON_180　EPSON系列打印机180×180dpi图象打印类
EPSON_60　EPSON系列打印机60×60dpi图象打印类
HP_600　HP系列打印机600dpi图象打印类
HP_300　HP系列打印机300dpi图象打印类
HP_150　HP系列打印机150dpi图象打印类
HP_100　HP系列打印机100dpi图象打印类
HP_75　HP系列打印机75dpi图象打印类

基类：PRINTER、EPSON_PRINTER、HP_PRINTER

说明：用于将图象缓存区中的图象输出到打印机上。一个类对应于一种打印机的一种分辨率，定义一个类对象，即可用该对象进行在相应打印机上相应分辨率下的图象打印。一个图象打印类对象必须与一个图形打印缓存类对象相连接，以使其能从一定的图象缓存区中读取图象数据。其也可与一个图形模式类对象或漫游模式类对象相连接，这时就将从显示存储器中读取图象数据，从而实现屏幕硬拷贝。

数据成员：

无

成员函数：

构造函数(VGABASE *v)

与一个图形打印缓存类对象、图形模式类对象或漫游模式类对象相连接。其在定义对象时被自动调用。

virtual void print(void)

进行图象打印。

virtual void outpaper(void)

换页(EPSON打印机)或出纸(HP打印机)

void nextrow(void)

按字符行距走一行纸。

15.2.6　事件类

数量：1

类名：Event

基类：无

说明：用于获取各类事件，并建立一个能保存各类事件的数据结构。定义该类的一个对象，即可用该对象进行事件的获取和保存。

数据成员：

unsigned what

事件类型。对事件类型的定义详见13.4.1.1节。

int key

发生键盘事件时，所按键的键码。

unsigned char mouse.buttons

发生鼠标事件时，鼠标三个按钮的状态。

int mouse.x, mouse.y

发生鼠标事件时，鼠标光标的位置。

int mouse.keystate

发生鼠标事件时，Ctrl、Alt、左Shift、右Shift四个键的状态。

unsigned message.command

发生消息事件时，消息类型的代码，该代码由应用程序自己定义。

void far *message.infoPtr

long message.infoLong

unsigned message.infoWord

int message.infoInt

unsigned char message.infoByte[2]

char message.infoChar[2]

用各种数据格式所表示的消息，对一个具体的消息来说，只有其中一种数据格式有效。

成员函数：

void getevent(void)

获取事件。该函数不等待事件的发生，如果没有事件发生，其就返回一个空事件。

void getevent(unsigned mask)

获取由mask所指定的一类或多类事件。该函数将等待所指定事件的发生。

void clear(void)

清除事件。

void postmessage(void)

发送一条消息。

void clearpostbox(void)

清除消息队列。

15.2.7 鼠标操作类

数量：1

类名：MOUSE

基类：无

说明：用于生成鼠标事件，并为应用程序提供鼠标操作的接口。由于系统中只会存在一个鼠标，因此该类的所有成员都被说明为静态的，不需要实例化即可使用它的各个成员。

数据成员：

static int OK

鼠标是否可用的标志。1表示鼠标可用，0表示不可用。

成员函数：

static void getevent(Event& ev)

生成并返回鼠标事件，所返回的事件放在ev中。该函数主要由事件类来调用，应用程序一般不需要直接调用它。

static int init(void)

初始化鼠标。在使用鼠标之前，应用程序必须运行该函数。

static void hide(void)

隐藏鼠标光标。

static void show(void)

打开鼠标光标。鼠标初始化之后，鼠标光标处于隐藏状态，应用程序需要调用此函数才能使鼠标光标显示出来。

static void getxy(int &x,int &y)

获得鼠标光标的当前位置。

static void putxy(int x,int y)

设置鼠标光标的位置。

static void putcentre(void)

将鼠标光标设置到屏幕中央。

static int inarea(int x1,int y1,int x2,int y2)

判别鼠标光标是否处于所给出的矩形区域内，如是则返回1，否则返回0。

static void setregion(int x1,int y1,int x2,int y2)

将鼠标光标的活动范围限制在所给定的矩形区域内。

static void restoreregion(void)

将鼠标光标的活动范围恢复为全屏幕。

static void setspeed(int n)

设置鼠标速度。缺省值为8，最快值为1。

static void setdoubletime(int n)

设置双击事件的最大时间间隔，单位为1/18.2s，缺省值为7。

static void setautotime(int n)

设置持续按下事件的延迟时间，单位为1/18.2s，缺省值为7。

15.2.8　鼠标光标显示类

数量：4

类名：MouseMap16　　16色及256色模式下的光标显示类

　　MouseMap256　256色模式下的光标显示类

　　MouseMapHigh　高彩色模式下的光标显示类

　　MouseMap16M　真彩色模式下的光标显示类

基类：MouseMap

说明：用于显示鼠标光标。一个类对应于一种光标图象，一种光标图象往往只适用于一定的色彩模式及一定范围的屏幕分辨率。鼠标光标的显示和维持由鼠标操作类来自动完成，而不需要应用程序在这方面做任何工作，应用只需定义一个光标显示类对象，并在不需要

的时候删除它。

数据成员：

无

成员函数：

构造函数(VGABASE *v)

将其自身连接到鼠标操作类中，以使鼠标操作类能用它来进行光标显示。同时，将一个图形模式类对象连接进来，以能使用各种图形显示功能来画出光标。其在定义对象时被自动调用。

析构函数

切断与鼠标操作类的联系，使鼠标操作类不在用它来进行光标显示。其在删除对象时被自动调用。

void setup(void)

将其自身连接到鼠标操作类中。其是构造函数的一部分，该函数主要在来回变换光标图象时使用。

15.2.9 键盘模拟鼠标函数

这是类库中仅有的两个独立的函数，它们不属于任何类。这两个函数用于安装和删除键盘模拟鼠标驱动程序。

void setup_key_mouse(void)

安装键盘模拟鼠标驱动程序。只有在真实的鼠标驱动程序不存在时，才会真正进行安装。该函数应在调用MOUSE::init()之前被调用。

void del_key_mouse(void)

删除键盘模拟鼠标驱动程序。如果在程序中运行了安装函数，在程序退出前就必须运行该函数。

15.2.10 漫游模式类

数量：11

类名：_640_480_16_roam 640×480×16色漫游模式类
_800_600_16_roam 800×600×16色漫游模式类
_1024_768_16_roam 1024×768×16色漫游模式类
_640_400_256_roam 640×400×256色漫游模式类
_640_480_256_roam 640×480×256色漫游模式类
_800_600_256_roam 800×600×256色漫游模式类
_1024_768_256_roam 1024×768×256色漫游模式类
_320_200_32k_roam 320×200×32K色漫游模式类
_512_480_32k_roam 512×480×32K色漫游模式类
_320_200_64k_roam 320×200×64K色漫游模式类
_512_480_64k_roam 512×480×64K色漫游模式类

基类：ROAM、VGABASE、VGA16、VGA256、VGAHIGH、VGA32K、VGA64K

说明：用于屏幕漫游，并在漫游状态下进行屏幕显示。一个类对应于一种漫游模式，每种

漫游模式都是在一种正常的图形模式的基础上建立。漫游模式类即继承于屏幕漫游基类，也继承于图形模式类的有关基类，因此它即具有屏幕漫游的功能，也具有所有的图形显示功能。定义某个漫游模式类的一个对象，即可通过该对象调用各屏幕漫游功能和图形显示功能。可以将一个漫游模式类对象连接到字符显示类、图象打印类、鼠标光标显示类中，从而能在漫游模式下进行字符显示、屏幕硬拷贝、鼠标操作。

数据成员：

其具有图形模式类的所有数据成员，其中WIDE和HIGH表示的是虚屏的大小。其增加了如下几个数据成员。

static int orgWide,orgHigh

实际屏幕的宽度和高度。

static int activHigh,fixedHigh

实际屏幕上，活动屏幕的高度及固定屏幕的高度。

static int begx,begy

实际屏幕的左上角在虚屏中的当前位置。

static int endx,endy

实际屏幕的右下角在虚屏中的当前位置。

成员函数：

其具有相应图形模式下的所有成员函数，同时增加了如下一些函数。

构造函数(int vwi,int vhi,int fhi=0)

接收应用程序所指定的虚屏的宽度vwi、高度vhi及进行屏幕分割时固定屏幕的高度，同时设置漫游模式的有关参数。

static void roam(int dx,int dy)

移动屏幕，dx为横向移动量，dy为纵向移动量。

static void setbxy(int bx,int by)

设置屏幕左上角在虚屏中的位置。

static void setexy(int ex,int ey)

设置屏幕右下角在虚屏中的位置。

static void autoroam(int x0,int y0,int xn,int yn)

自动移动屏幕以使虚屏中的一个矩形区域能处于实际屏幕之中。(x0,y0)为矩形区域的左上角在虚屏中的位置，xn，yn分别为矩形区域的宽度和高度。

void setup_key_roam(int d=8)

安装键盘自动漫游程序，d为每次按键使屏幕移动的量。键盘模拟鼠标程序和键盘自动漫游程序不能同时安装。

void del_key_roam(void)

删除键盘自动漫游程序。如果应用程序安装了键盘自动漫游程序，就必须在退出前删除它。

15.3　实用程序使用说明

本书共提供了5个与图形有关的实用程序，这里介绍它们的使用方式。

15.3.1 VESA图形模式检测程序

执行程序文件名为：TVESA.EXE，源程序文件名为TVESA.CPP，存放在\VGA\TVESA目录中。

程序运行后，将检测出当前显示卡所支持的所有VESA显示模式，包括图形模式和字符模式，然后检测出每种模式下的有关参数及显示存储器的分页情况。其一屏显示一种模式的情况，用“PgUP”和“PgDn”键前后翻页，以查看各显示模式的情况，按“Esc”键退出程序。

下面列出由该程序所输出的两种显示模式的情况，括弧中的信息为对每一行的说明，它们不是程序输出的。

```
VESA Version 1.2  (VESA版本号)
Copyright 1988-1991 TRIDENT MICROSYSTEMS INC.  (供应商信息)
Number of VESA Mode : 23 -- 16  (共支持23种VESA模式，当前显示的是第16种)
VESA Mode No. :  104H  (VESA模式号为104H)
Graphics Mode   1024 * 768 * 16  (为图形模式，分辨率为1024×768，颜色数为16)
Win A: Write Enable    Read Enable  (窗口A可写、可读)
Win A Segment : A000  (窗口A段地址)
Win B: None  (窗口B没有)
Win B Segment : A000  (窗口B的段地址，无效)
Window Granularity : 64K  (窗口粒度)
Page Size : 64K  (页尺寸)
Bytes of scanline : 128Byte  (一条扫描线在一个位面中所占字节数)
Display Memory Size : 393216Bytes    384K  (所占显示存储器容量)
Number of Planes : 4  (位面数)
Bits of Pixel : 4  (每个象素点所占位数)
Display Memory model :  4 Planes  (显示存储器组织模式)
Select Page Function Ptr. : C000:187A  (换页功能的地址)
```

该程序所检测到的每一种图形模式并不一定都为当前的显示卡或显示器所支持，主要是那些高彩色模式、真彩色模式及分辨率超过1024×768的模式会出现这一问题。所检测出的一种模式是否真正被当前硬件所支持，只有通过使用这一模式才能确认，本书所提供的演示程序中的图形模式演示功能可以用来进行这种测试。

如果当前显示卡不支持VESA标准，程序将给出这一信息并退出。

15.3.2 图形显示速度测试程序

执行程序文件名为：VS.EXE，源程序文件名为：VS.CPP，存放在\VGA\SPEED目录。

该程序用于测试各种图形模式下的图形显示速度，测试的方式详见本书的10.3节。该程序可用于对硬件的性能进行测试和比较。

该程序的运行过程中不需要作任何操作，其自动依次对各图形模式进行测试，测试完后在屏幕显示出测试结果，其对每种图形模式给出6个速度：DS1、DS2、DS3和LS1、LS2、LS3，前三个为写点操作的速度，后三个为画扫描线操作的速度，其中，DS1、LS1为单点

速度，单位为每秒所写千个象素点数；DS2、LS2整屏速度，单位为每秒所写满的屏幕数；DS3、LS3为数据传输率，单位为每秒K字节数。

如果硬件不支持某种图形模式，就将跳过该模式，所输出的该模式的速度就都为0。

该程序的运行时间较长，在某些机器上可能会达到十几分钟，如果想中途退出，可按“Esc”键，这是其仍将输出已测试完的图形模式的测试结果。

15.3.3　小汉字库构造程序

执行程序文件名为：XHZK.EXE，源程序文件名为：XHZK.CPP，存放于\VGA\XHZK目录。

该程序用于从一个正常的汉字库中抽取出若干字符，以构造出一个小字库，其只能用于构造16×16、24×24、32×32、36×40的点阵小汉字库及UCDOS的矢量小汉字库。其所构造出的小汉字库能直接被本书所提供的字符显示程序使用。

其使用方式参见本书的11.3节。

15.3.4　大字打印程序

执行程序文件名为：PLC.EXE，源程序文件名为：PLC.CPP，存放于\VGA\PLC目录。

该程序用于打印单个的大汉字及汉字图形字符，其所能打印的字符的大小仅受打印机幅宽的限制。该程序只能用于EPSON系列和HP系列的打印机，其采用了这两种打印机上的最低分辨率，以提高速度，并减少对内存或硬盘空间的要求。其只能使用UCDOS的矢量字库，所打印出的为空心汉字。

用命令行参数对程序进行，下面给出其命令行参数的格式：

```
PLC printer_type size C_chart HZK_filename
```

其中：printer_type　表示打印机类型的字符串，只能是“EPSON”或“HP”

size　所要打印出的字符的大小，以厘米为单位

C_chart　一个汉字字符或汉字图形字符

HZK_filename　字库文件名及路径名，只能是UCDOS的矢量汉字库

使用示例：

```
PLC EPSON 40 典 hzkslktj （打印一个40厘米的楷体“典”字）
PLC HP 20 《 c:\ucdos\hzkslt （打印一个20厘米的汉字图形字符“《”）
```

由于要在命令行输入汉字，所以该程序需要在汉字系统下使用，但某些汉字系统屏蔽了正常的打印控制指令，而无法在其下进行图形打印，这时就可先在汉字系统下建立一个包含有相应命令的批处理文件，然后在英文状态下运行该批处理文件。

15.3.5　演示程序

执行程序文件名为：VGADEMO.EXE，其对应有6个源程序：VGADEMO.CPP、DMODE.CPP、DFONT.CPP、DPRT.CPP、DMOUSE.CPP、DROAM.CPP，这些文件都存放在\VGA\DEMO目录中。

该程序对VGAL.LIB类库所提供的一些主要功能进行演示，包括：图形模式演示、字符显示演示、图形打印演示、鼠标演示及屏幕漫游演示。

图形模式演示在18种图形模式下显示一些能反映模式的色彩情况及分辨率情况的图

形，该演示功能可用来检测硬件对各种图形模式的支持情况。

字符显示演示在屏幕上显示本书所支持的所有字库的字符。这项演示功能需要在硬盘的\VGA\FONT目录中存放有各种字库，如某些字库不存在就不能显示相应的字符，但这不影响其它的演示，也不影响程序的运行。

图形打印演示在EPSON和HP打印机上以各种分辨率进行图形打印。

鼠标演示在4种色彩模式下显示相应的鼠标光标图象，并进行鼠标事件测试。如果系统中没有安装鼠标驱动程序，就将使用键盘模拟鼠标，这时，用键盘中间的4个方向键来控制鼠标的移动，用左Alt、左Ctrl和左Shift来模拟鼠标的左、右、中三个按钮。鼠标事件测试要用到\VGA\FONT目录中的ASCK16字库。

屏幕漫游演示用800×600的屏幕来漫游一个1024×1024的虚屏，其分别使用键盘和鼠标来进行屏幕自动漫游。

程序用一个简单的二级菜单进行操作，在进入某个演示项目后，按任意键或“Esc”键即可退出或进入下一步。

参 考 文 献

[1] 王知珩.计算机图形显示技术.哈尔滨：哈尔滨船舶工程学院出版社，1993
[2] 来文占等.微机显示技术实用手册.北京：电子工业出版社，1992
[3] 陈振初，蔡宣平.计算机图形显示原理(软件).长沙：国防科技大学出版社，1991
[4] 王东泉.计算机图形学与CAD技术.上海：上海交通大学出版社，1992
[5] 求伯君.深入DOS编程.北京：北京大学出版社，1993