第 1 章　简介

1.1　计算机绘图及图象处理的应用

目前计算机绘图及图象处理的应用相当广泛，从电子游戏到太空探测，几乎在各行各业都可以见到，以下我们就介绍几个常见的应用领域：

娱乐事业

各式大型电动玩具以及计算机游戏，不论是角色扮演、动作冒险、战略、模拟、益智、射击……等，都涉及到颇复杂的图象处理。

刑事侦查

刑警在办案时，必须将照片、指纹、笔迹、人像等案情相关资料，作进一步的处理和鉴别，找到证据或线索。

医学研究

借助计算机绘图及图象处理技术，如断层扫描、超声波成象、X 射线成象等，可以让医生更快更准确地诊断疾病。

科学研究

诸如飞机航道显示、自然灾害预测、宇宙星体照片等，都是利用计算机绘图及图象处理来取得所需信息，当然每天天气预报中的卫星云图，也是应用的范畴。

工业生产

大家熟悉的计算机辅助设计(Computer Aided Design，简称 CAD)和计算机辅助制造(Computer Aided Manufacturing，简称 CAM)，在工业生产中广泛应用是众所周知的，越来越多的人已不再用手去画建筑设计图或汽车结构图。

除了上述应用之处，还有许许多多的应用实例出现在我们的生活之中，例如：电子书籍、电子相薄、电子暗房以及各式各样的多媒体应用系统。电子暗房就是图象处理软件。

最近几年由于 PC 的速度愈来愈快，价格愈来愈低，加上各种 Super VGA 卡的产生，当然最主要的原因还是由于 Windows 的出现且日趋成熟，使我们越来越难发现有 Macintosh 计算机能提供、而 PC 却做不到的事情。过去 Macintosh 的图象处理软件遥遥领先，Adobe Photoshop for Mac 使得 PC 上的图象处理软件相形见拙。但随着 Photoshop 与 Fractal Design 公司的 Color Studio 相互配合且一起进入 Windows 后，现在的形势已有所逆转。还记得几年前，能够在 Windows 上面处理全彩图象的软件可以说是屈指可数，不过在图象处理软件厂商几年来的细心耕耘之下，已经呈现出崭新局面，而今角逐此间市场的各路英雄已不在少数，以下即为常见的图象处理软件开发厂商及其软件名称：

Adobe System Inc.	Adobe Photoshop for Windows
Aldus Corp.	PhotoStyler
Micrografx Inc.	Picture Publisher

	PhotoMagic for Windows
Computer Associates	CA－Cricket Paint
	CA－Cricket Image
Image－In	Image－in－Color

(注:Adobe 已于 1994 年 3 月以 525 百万美元的高价买下 Aldus。)

PC 的图象处理软件市场规模还不能与 Macintosh 环境的相提并论,但是它发展得相当迅速,我们相信当 24 位(bit)图形及图象日渐普及于现在的计算机世界时,图象处理软件将会是主流产品之一,而在不久的将来,计算机绘图及图象处理的应用也会更深入我们的生活。

1.2 相关知识简介

介绍完目前有关图象处理方面的应用后,您是否也想要投入这众所瞩目的领域呢?首先我们必须先来了解有关图象文件的基本概念。

我们首先了解一下如何储存那些丰富多彩的计算机展示画面呢?原则上这些图象文件大致可分为三大类:

点阵型

采用位映射(bit－mapped)的方式储存图形图象。例如大家最熟悉的中文字型,它的 15＊16 字型和 24＊24 字型就是使用此种方式存储。一般我们所常见的点阵型图象文件常用 BMP、PCX 、GIF 等作后缀,而这些图象文件也就是本书所要详细探讨的内容。

向量型

采用直线(line)、曲线(curve)、矩阵(rectangle)、圆形(circle)等向量(vector)的方式储存图形。市面上所谓的向量字型就是使用这种方式储存(当然有的向量图内也可以储存点阵型图),而一般我们所常见的向量图文件有 Windows 的.WMF、Lotus 的.PIC、CorelDraw 的.CDR 等。

动画型

所谓动画型图象其实可以说是由许多的点阵型图象所组成,但是有些更高级的动画型图象则可以分析出两个画面的差异,然后只储存其有差异部分。例如有一只海鸥正飞过一座高山,此时当作背景的高山是不会有任何改变的,因此只需储存画面间海鸥飞过时所改变的地方即可。一般我们常见的动画图象文件有 Animator 的.FLI、MacroMind Director 的.MMM 等,而目前最受众人瞩目的动画型图象则为 MPEG(Motion Picture Experts Group)数字电视图象。

每一种类型的图形图象文件都有很多种存储格式,这些图形图象文件储存格式的提出,可能是以下几种情形:

为配合软件或硬件而制定

如 ZSoft 的 Paintbrush 所使用的.PCX、Microsoft Windows 所使用的.BMP 以及 Truevision Targe 全彩显示卡所使用的.TGA。

计算机相关厂商制定

如由 Aldus、Microsoft、Hewlett－Packard、Dest、DataCopy、Microtek、Media Cybernetics、New Image、Technology 和 Software Publishing 八家公司共同发表的 TIFF。

国际性的组织或协会所制定

如由 ISO 和 CCITT 为静态图象所建立的第一个国际数字图象压缩标准——JPEG。

在正式介绍这些图象文件格式之前，我们必须先了解图象数据的结构。

图象数据事实上就是一个由许多元素(element)组成的二维矩阵，由于矩阵中每个元素相当于该图中的每一点(dot)，因此它们被称为象素(pixel)，而图象数据的色彩数目(color number)则是由每点所需位数(bits per pixel)来决定，其计算方式如下：

$$\text{Color Number} = 2^{(\text{Bits Per Pixel})}$$

Bit Per Pixel	Color Number
1－bit	2－color
4－bit	16－color
8－bit	256－color
16－bit(15－bit)	high－color
24－bit	true－color
32－bit	true－color

由于彩色计算机屏幕所显示的每种颜色都是以 RGB(Red、Green、Blue)三基色组成，因此大多数的图象文件在储存彩色图象数据时，都是采用 RGB 色彩系统来描述颜色，彩色图象大致可以分为三组：

真彩色图象

如果图象数据中每个象素的颜色直接以 RGB 来表示，其中 R、G、B 各占 1 个字节(即每个象素占 3 个字节)，则该图象称为真彩色(true color)图象，也称全彩色(full color)图象。以一张 640×480 的 24 位的真彩色图象为例，其所需的储存空间为：

$$640 \times 480 \times 3 \approx 900\text{KB}$$

深彩色图象

如果图象数据中每个象素的颜色值仍是直接以 RGB 来表示，但每个象素仅占 2 个字节(亦即 R.G.B 可能各占 5 个位，也可能某一种颜色占了 6 个位)，该图象就称为深彩色(high color) 图象。以一张 640×480 的 16 位的深彩色图象为例，其所需的储存空间为：

$$640 \times 480 \times 2 \approx 600\text{KB}$$

伪彩色图象

如果图象数据中各种颜色的 RGB 值储存在彩色表(color table)或调色板(palette)内，而图象资料中每个象素的颜色值为指向调色板的索引值(index)，则该图象称为含调色板(palette color)图象，通常称为伪彩色(pseudo color)图象。由于图象资料中每个象素的颜色值以不超过 1 个字节为原则(通常每个象素占 4 个位或 8 个位)，因此调色板中最多可以储存 256 种颜色(通常为 16 种或 256 种颜色)。以一张 640×480 的 8 位的伪彩色图象为例，其所需的储存空间为：

$$640 \times 480 + 256 \times 3 \approx 300.75\text{KB}$$

大部分的图象文件格式都支持 24 位的真彩色图象，但是支持 32 位的图象文件格式

的就不多了，以下为我们介绍的六种图象文件格式及所支持的每象素点位数如下所示：

图象文件格式	1 位	4 位	8 位	16 位	24 位	32 位
BMP	*	*	*		*	
JPEG			*		*	
PCX	*	*	*		*	
TGA			*	*	*	*
GIF	*	*	*			
TIFF	*	*	*		*	*

至于图象文件在储存灰度图象(gray scale image)数据时，也是采用 RGB 色彩系统来描述颜色，只不过 R.G.B 三种颜色值都相同(即 R=G=B)罢了，而其图象数据描述颜色的方式，则可以是上述三类彩色图象数据中的一类(通常是以 8 位的方式来储存数据)。虽然需要使用灰度图象的人不在少数，但是并非所有图象文件格式都支持灰度图象，即能够从文件结构中直接判断是否为灰度图象，以下为我们介绍的六种图象文件格式，其支持的情形如下所示：

图象文件格式	支持灰度图象
BMP	
JPEG	*
PCX	*
TGA	*
GIF	*
TIFF	*

第 2 章　PCX 图象文件

2.1　简介

PCX(也称 PCC)是随 zSoft 公司开发出的 Paintbrush 绘图软件公布的图象文件格式。由于 Paintbrush 过去曾经是 Microsoft mouse 的附属品(bundled with the mouse),因此它早已成为众所皆知的绘图软件,而其所使用的 PCX 图象文件,也因为具有简单易懂且编程容易的图象压缩方法,几乎每个支持图象的应用软件,都能够处理 PCX 图象文件格式。

虽然 PCX 图象文件广泛地为大家接受并使用,但其缺乏远见的图象文件结构也常让人感到不便(在下一节我们将会对 PCX 文件结构作更进一步的说明),不过随着 Paintbrush 功能的增强,PCX 的版本也有所更新,因此已能支持真彩色图象的 PCX,仍将在众多的图象文件格式中占有一席之地。

2.2　PCX 文件结构

PCX 文件结构十分简单,仅分成文件头和图象压缩数据两个部分(如果是 8 位的 PCX 图象文件,则还有 256 色调色板数据储存于文件尾)。下面我们将分别作详细的说明:

2.2.1　文件头信息

PCX 的文件头全部占有 128 字节,其数据结构如下:

```
typedef struct pcxheader
{
    BYTE byManufacturer;
    BYTE byVersion;
    BYTE byEncoding;
    BYTE byBits;
    WORD wLeft;
    WORD wTop;
    WORD wRight;
    WORD wBottom;
    WORD XResolution;
    WORD wYResolution;
    BYTE byPallette[48];
    BYTE byReserved;
    BYTE byPlanes;
```

```
    WORD wLineBytes;
    WORD wPaletteType;
    WORD wScrWidth;
    WORD wScrDepth;
    BYTE byFiller[54];
  } PCXHEADER;
```

byManufacture

PCX 识别码。固定为 0×0A,使用者可根据这个域来判断是否为 PCX 图象文件。

byVersion

表示使用的 Paintbrush 版本。其值代表的意义如下:

0: PC Painbrush 2.5

2: PC Painbrush 2.8(包含调色板数据)

3: PC Painbrush 2.8(未包含调色板数据)

4: PC Painbrush for Windows

5: PC Painbrush 3.0、PC Paintbrush Ⅳ、Ⅳ Plus、PC Painbrush Plus for Windows、Publisher's Paintbrush 或更新的版本。

而 Paintbrush 各版本与其处理色彩的能力如下:

byVersion	单色	4 色	8 色	16 色	256 色	全彩色
0	*	*				
2	*	*	*	*		
3	*	*	*	*		
4	*	*	*	*		
5	*	*	*	*	*	*

倘若 byVersion 的值为 3,则表示使用系统缺省的调色板数据。

byEncoding

数据压缩方式。其值目前固定为 1,表示采用 Runlength 压缩法。倘若未来 PCX 加入新的数据编码方法,则需根据此域值来判断数据的压缩方式。

byBits

每个象素所需的位数(指各个色彩平面)。

wLeft

图象相对于屏幕(screen)左上角的 X 坐标(以象素为单位)。

wTop

图象相对于屏幕(screen)左上角的 Y 坐标(以象素为单位)。

wRight

图象相对于屏幕(screen)右下角的 X 坐标(以象素为单位)。

wBottem

图象相对于屏幕(screen)右下角的 Y 坐标(以象素为单位)。

图象宽度及高度的计算方式如下:

宽度 = wRight − wLeft + 1

高度 = wBottom − wTop + 1

wXResolution

图象的水平分辨率(每英寸有多少个象素)。

wYResolution

图象的垂直分辨率(每英寸有多少个象素)。通过 wXResolution 和 wYResolution 可以使图象在输出设备上有最佳的效果。

byPalette

调色板数据。由于 header palette 仅有 48(16×3)字节,因此最多只能储存 16 种颜色的调色板数据,其数据(RGB)储存方式如下:

RGBRGB…RGB

读者该已想到:"PCX 是如何储存 256 色图象的调色板数据呢?"事实上,以其原有文件的结构框架确实无法处理 256 色的图象(48 字节的空间怎么可能储存 768 字节的数据呢?),因此图象文件的最后,并以 0×0C 为其识别码如下所示:

PCX 文件结构	字节数
文件头信息	128
图象压缩数据	-
0×0C	1
256 色调色板	768

此时 header palette 的数据便毫无用处(浪费了 48 字节的空间),而这也就是为什么我们前面曾经提及其缺乏远见的文件结构而让人不便的原因了。

byReserved

保留未用。此域设定为 0。

byPlanes

图象的色彩平面数。

wLineBytes

图象的宽度(以字节为单位)。此值必须是偶数(即 2 的倍数)。

wPalatteType

调色板类型,其值代表的意义:

1:彩色或单色

2:灰度

wScrWidth

制作此图象文件的屏幕宽度(以象素为单位)。此值以 0 为基准(base on 0),也就是说此值为实际的屏幕宽度减 1。

wScrDepth

制作此图象文件的屏幕高度(以象素为单位)。此值以 0 为基准(base on 0),也就是说此值为实际的屏幕高度减 1。

byFiller

保留(留待新版扩充)。此域应设定为0。

2.2.2 图象压缩数据

PCX图象文件的数据虽是经过压缩后才储存(在下一节我们会对PCX压缩方法作进一步的解说),但因为图象原始数据的排列方式与色彩数目有极大的关系,而这也涉及到图象压缩数据的储存方式,所以我们必须对此问题有更深的了解。在PCX图象文件中色彩数目是由文件头内的byBits和byPlanes两个域决定的,其常见的组合方式如下:

byBits	byPlane	色彩数目
1	1	单色
1	4	16色
8	1	256色
8	3	全彩色

下面我们将分别就上述常见的图象数据排列、储存方式作详细的说明:

单色图象

单色图象的每个象素都是以1个位来表示(即每8个象素组成1个字节),并且按照由左到右、由上到下的排列顺序来处理。例如,一个宽为22象素的单色图象,则其数据储存于存储器中的情形如下:

单色图象 ┌──────22位──────┐

scan line 0 位0—位7 位8—位15…

scan line 1 位0—位7 位8—位15…

⋮ ⋮ ⋮

	Bit No.								
byte 0	0	1	2	3	4	5	6	7	scan line 0
byte 1	8	9	10	11	12	13	14	15	scan line 0
byte 2	16	17	18	19	20	21	–	–	scan line 0
byte 3	–	–	–	–	–	–	–	–	scan line 0
byte 4	0	1	2	3	4	5	6	7	scan line 1
byte 5	8	9	10	11	12	13	14	15	scan line 1
byte 6	16	17	18	19	20	21	–	–	scan line 1
byte 7	–	–	–	–	–	–	–	–	scan line 1
⋮	⋮	⋮							

"–"表示值为0的位

倘若每条水平扫描线(scan line)的数据,其最后一个字节的位数未满8(即所有的位数不为8的倍数),则必须补上填充位(fill bits).由于PCX规定每条scan line的字节数必须是偶数(请参考文件头内的wLinBytes域说明),因此在上述的例子中,每条scan line的结尾都必须加上10个填充位。

16 色图象

16 色图象的每个象素都是以 4 个位来表示(即每 2 个象素组合成 1 个字节),且按照由左到右、由上到下的排列顺序来处理。因为 PCX 将 4 个位分别对应到 4 个 plane 中(对应顺序为 plane3、plane2、plane1、plane0),所以我们可以把每个 plane 都看作是一张单色图象,而每条 scan line(指原始图象)的正确数据则必须由四张单色图象(即 4 个 plane)的 scan line 数据组合才能得到。例如,一个宽度为 11 象素的 16 色图象,其储存于存储器中的情形如下图 2-1:

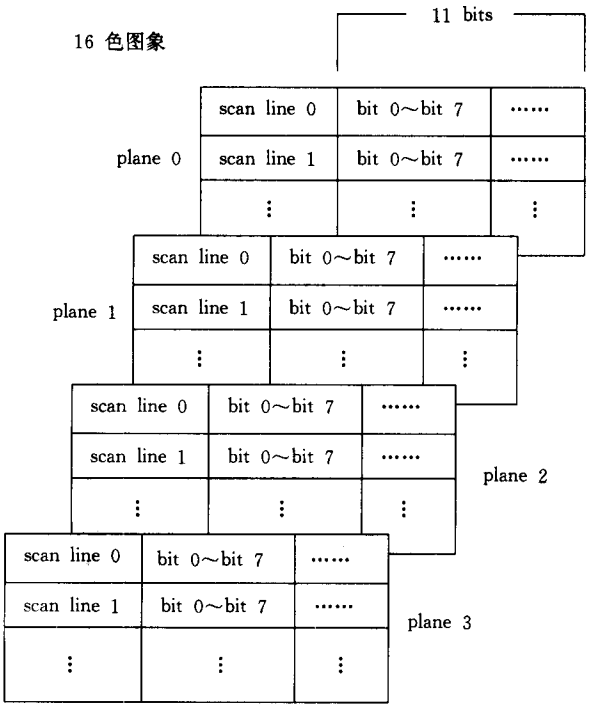

图 2-1

Bit No.

byte 0	0	1	2	3	4	5	6	7	┐
byte 1	8	9	10	–	–	–	–	–	┘ – scan line 0 (p3)
byte 2	0	1	2	3	4	5	6	7	┐
byte 3	8	9	10	–	–	–	–	–	┘ – scan line 0 (p2)
byte 4	0	1	2	3	4	5	6	7	┐
byte 5	8	9	10	–	–	–	–	–	┘ – scan line 0 (p1)
byte 6	0	1	2	3	4	5	6	7	┐
byte 7	8	9	10	–	–	–	–	–	┘ – scan line 0 (p0)
byte 8	0	1	2	3	4	5	6	7	┐
byte 9	8	9	10	–	–	–	–	–	┘ – scan line 1 (p3)

byte 10　0 1 2 3 4 5 6 7 ┐– scan line 1 (p2)
byte 11　8 9 10 – – – – – ┘
⋮　　⋮　　⋮
"–"表示值为 0 的位
(pX)表示第 X 个 Plane

由于我们可以把每个 plane 都看作是一个单色图象，因此在上述的例子中，每条水平扫描线(指每个 plane)的结尾都必须加上 5 个填充位。而在上例中，由 4 张单色图象(即 4 个 plane)的水平扫描线数据组合所得的每条水平扫描线及原始图象的正确数据如下：

scan line 0 (Bit No. in Plane X)

byte 0	0(p3)	0(p2)	0(p1)	0(p0)	1(p3)	1(p2)	1(p1)	1(p0)
byte 1	2(p3)	2(p2)	2(p1)	2(p0)	3(p3)	3(p2)	3(p1)	3(p0)
byte 2	4(p3)	4(p2)	4(p1)	4(p0)	5(p3)	5(p2)	5(p1)	5(p0)
byte 3	6(p3)	6(p2)	6(p1)	6(p0)	7(p3)	7(p2)	7(p1)	7(p0)
byte 4	8(p3)	8(p2)	8(p1)	8(p0)	9(p3)	9(p2)	9(p1)	9(p0)
⋮				⋮				

(pX)表示第 X 个 Plane

256 色图象

256 色图象的每个象素都是以 8 个位来表示(即每 1 个象素就是 1 个字节)，且按照由左到右、由上到下的排列顺序来处理。事实上，除了表示每个象素的位数有所差异之外，其结构上可以说是和单色图象相同，而图象数据是调色板数据的索引值。例如，一个宽度为 7 个象素 256 色图象，其数据储存在内存中的情形如下：

256 色图象

scan line 0	象素 0—象素 6	
scan line 1	象素 0—象素 6	
⋮	⋮	
	Pixel NO	
byte 0	0	scan line 0
byte 1	1	scan line 0
byte 2	2	scan line 0
byte 3	3	scan line 0
byte 4	4	scan line 0
byte 5	5	scan line 0
byte 6	6	scan line 0
byte 7	–	scan line 0
byte 8	0	scan line 1
byte 9	1	scan line 1

byte 10　　2　　scan line 1

⋮　　⋮　　⋮

“－”表示值为 0 的字节(byte)

在上例中，每条水平扫描线的结尾都必须加上 1 个填充字节(即 8 个填充位)。

全彩图象

全彩图象的每个象素都是以 24 位来表示，即每 1 个象素是由 3 个分别代表 RGB 的字节组成。并按照从左到右、由上到下的排列顺序来处理。因为 PCX 将 3 个字节分别对应到另 3 个 plane 中(对应顺序为 plane2, plane1, plane0)，所以我们可把每个 plane 都看作是一张 256 图象，而每条 scan line(指原始图象)的正确数据则必须经由三张 256 色图象(即 3 个 plane)的 scan line 数据组合才能得到。举例来说，如果有一宽度为 3 象素的全彩色图象，其数据存在存储器中的情形如下图 2-2 所示：

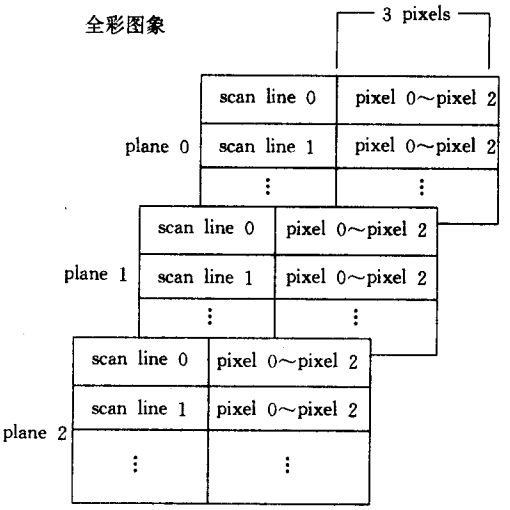

图　2-2

Byte No.			in Plane X	Color		
0	1	2	－	R	Plane2	┐
0	1	2	－	G	Plane1	├ scan line 0
0	1	2	－	B	Plane0	┘
0	1	2	－	R	Plane2	┐
0	1	2	－	G	Plane1	├ scan line 1
0	1	2	－	B	Plane0	┘
⋮	⋮	⋮		⋮		

“－”表示值为 0 的字节(byte)

由于我们可以把每个 plane 都看作是一张 256 色图象，因此在上例中，每条 scan line

(指每个 plane)的结尾都必须加上 1 个填充字节(即 8 个填充位)。在上例中,由三张 256 色图象(即 3 个 plane)的 scan line 数据组合得到的每条 scan line (指原始图象)的正确数如下:

	Byte No.	Color	Pixel No.	
byte 0	0(p2)	R		
byte 1	0(p1)	G	0	scan line 0
byte 2	0(p0)	B		
byte 3	1(p2)	R		
byte 4	1(p1)	G	1	scan line 1
byte 5	1(p0)	B		

(pX)表示第 X 个 plane

2.3 PCX 压缩方法

PCX 图象文件是采用一种称为重复段长度(Run Length Encoding,简称 RLE)的压缩方法,它是先计算原始数据中各个数据的重复次数(count),然后用该数据的重复次数加上数据本身来取代原始数据,达到压缩的目的,其编码原则为:

* 图象数据是以字节为单位进行编码。

* 每条 scan line(指每个 plane)必须分开压缩。

* 若遇到重复的图象数据,其长度值为 iCount,则先存入资料长度值(iCount | 0×C0),再存入重复的图象资料。

* 若遇到不重复的图象数据则直接存入。如果该不重复的值大于 0×C0,则必须先存入资料长度值(1|0×C0),再存入该不重复的图象数据(即采用重复数据的方式存入),以避免被误认为数据长度值。

假设有一条 scan line 的数据如下:

0×09 0×09 0×09 0×09 0×09 0×13 0×F9 0×09 0×08 0×08 0×08

则其编码结果为:

0×C5 0×09 0×13 0×C1 0×F9 0×09 0×C3 0×08

* 由于 iCount 的值最大仅能为 63,所以如果(iCount>63),则必须分数次处理。

假设有一条 scan line 的数据为 132 个 0×98,则其编码结果为:

0×F F 0×98 0×FF 0×98 0×C6 0×98

为了能够分辨两种不同的储存方式(重复数据和不重复数据),PCX-RLE 以数据的最高两个位(位 7 和位 6)作为判断的依据:输出不重复数据时,其最高两位必为 0;输出重复数据时,数据长度值的最高位必为 1,这也就是为什么 PCX-RLE 限制 iCount 的值最大仅能为 63 的原因。

当我们了解 PCX-RLE 的编码原则之后,便可以推敲出 PCX-RLE 是如何解压缩的,以下即为 PCX-RLE 化解一条 scan line 的流程:

(1) 读入一个字节到 byChar

(2) if ((byChar & 0×C0) = =0×C0)

{

iCount = byChar & 0×3F;

读入下一个字节并将其重复 iCount 次;

}

else

{

直接读入下一个字节;

}

(3) 重复(1)和(2)的动作直到化解完一条 scan line。

PCX-RLE 压缩法比 LZW 压缩法及霍夫曼压缩法更为简单易懂、容易实现且执行速度快,不过在压缩比例上却略逊一筹,或许鱼与熊掌真不是那么容易可同时吃到的。

2.4 PCX 编程

以下即为读写 PCX 图象文件的程序清单。

<PCXMAIN>

```
all: pcxmain.exe

WINCC   = cl -c -Gsw -W3 -AM -Zp -Ow
WINLINK = link /NOD /align:16

.c.obj:
    $(WINCC) $*.c

OBJS = pcxmain.obj  \
       wndproc.obj  \
       imprtpcx.obj \
       pcxdrle.obj  \
       exprtpcx.obj \
       pcxerle.obj  \
       showmeta.obj \
       mcsutil.obj  \
       fileopen.obj \
       filesave.obj

pcxmain.res  : pcxmain.rc       sys.h    mcsdef.h    mcspcx.h
        rc -r pcxmain.rc

pcxmain.obj  : pcxmain.c        sys.h    mcsdef.h    mcspcx.h
      $(WINCC) pcxmain.c

wndproc.obj  : wndproc.c    sysextrn.h    mcsdef.h    mcspcx.h
```

```
        $(WINCC) wndproc.c

imprtpcx.obj  : imprtpcx.c  sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) imprtpcx.c

pcxdrle.obj   : pcxdrle.c   sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) pcxdrle.c

exprtpcx.obj  : exprtpcx.c  sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) exprtpcx.c

pcxerle.obj   : pcxerle.c   sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) pcxerle.c

showmeta.obj  : showmeta.c  sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) showmeta.c

mcsutil.obj   : mcsutil.c   sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) mcsutil.c

fileopen.obj  : fileopen.c  sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) fileopen.c

filesave.obj  : filesave.c  sysextrn.h    mcsdef.h    mcspcx.h
        $(WINCC) filesave.c

pcxmain.exe   : $(OBJS) pcxmain.def pcxmain.res
                $(WINLINK) @pcxmain.lnk,,,libw mlibcew,pcxmain
                rc pcxmain.res
```

<EXPRTPCX.C>

```
#include "sysextrn.h"
#include "memory.h"

int iExportPCX(LPSTR lpszSrcFName,LPSTR lpszDstFName);
int iCreatePCX(int iFile,int iTempFile,LPPCXC_VAR lpPCXCVar,WORD wWidthBytes);

WORD wRoundUp(WORD wA,WORD wB);
int iEncodeLine(int iTempFile,LPPCXC_VAR lpPCXCVar,LPSTR lpTemp);
void vStatusProcess(int iRatio);

int iExportPCX(LPSTR lpszSrcFName,LPSTR lpszDstFName)
{
   BITMAPFILEHEADER    BFH;
   BITMAPINFOHEADER    BIH;
   PCXHEADER           PCXH;
   PCXC_VAR            PCXCVar;
   RGBQUAD             TmpPal[256];
   WORD                wColors;
   WORD                wWidthBytes;
```

```
WORD                wTemp;
WORD                wi;
BYTE                szStr[256];
BYTE                byPCX_Pal[769];
int                 iFile;
int                 iTempFile;
int                 iReturn;

iFile = _lopen(lpszSrcFName, READ);

/* BMP File Header */
_lread(iFile, (LPSTR)&BFH, sizeof(BFH));

/* Is a Microcosm MetaFile? */
if ( ! CHECK_MMF(BFH.bfType) )
{
   _lclose(iFile);
   wsprintf(szStr,"非[MMF]文件%s",lpszSrcFName);
   MessageBox(ghParentWnd, szStr, NULL, MB_OK | MB_ICONSTOP);
   return 1;
}

/* BMP Info Header */
_lread(iFile, (LPSTR)&BIH, sizeof(BIH));

PCXCVar.wWidth     = (WORD)BIH.biWidth;
PCXCVar.wDepth     = (WORD)BIH.biHeight;
PCXCVar.wPlanes    = ( (BIH.biBitCount == 4) ? 4 :
                       (BIH.biBitCount == 24) ? 3 : 1 );
PCXCVar.wBits      = BIH.biBitCount;
PCXCVar.wLineBytes = ( (PCXCVar.wBits >= 8) ? wRoundUp(PCXCVar.wWidth,2) :
                                              WORD_WBYTES( PCXCVar.wWidth ) );

wWidthBytes = (WORD)DWORD_WBYTES( BIH.biWidth * (DWORD)PCXCVar.wBits );
wColors     = IMAGE_COLORS(PCXCVar.wBits, 1);

if ( wColors )
{
   /* Read Palette */
   _lread(iFile, (LPSTR)TmpPal, (wColors<<2));

   if ( PCXCVar.wBits == 8 )
   {
      byPCX_Pal[0] = 0x0C;
      wTemp = 1;
      for(wi = 0;wi<wColors;wi++)
      {
```

```
            byPCX_Pal[wTemp++] = TmpPal[wi].rgbRed;
            byPCX_Pal[wTemp++] = TmpPal[wi].rgbGreen;
            byPCX_Pal[wTemp++] = TmpPal[wi].rgbBlue;
        }
    }
    else
    {
        _fmemset(PCXH.byPalette,0,48);
        wTemp = 0;
        for(wi=0;wi<wColors;wi++)
        {
            PCXH.byPalette[wTemp++] = TmpPal[wi].rgbRed;
            PCXH.byPalette[wTemp++] = TmpPal[wi].rgbGreen;
            PCXH.byPalette[wTemp++] = TmpPal[wi].rgbBlue;
        }
    }
}
iTempFile = _lcreat(lpszDstFName,0);
/* Write The PCX File Header */
PCXH.byManufacturer = PCX_MARK;
PCXH.byVersion      = 5;
PCXH.byEncoding     = 1;
PCXH.byBits         = (BYTE)( (PCXCVar.wBits>=8) ? 8 : 1 );
PCXH.wLeft          = 0;
PCXH.wTop           = 0;
PCXH.wRight         = PCXCVar.wWidth - 1;
PCXH.wBottom        = PCXCVar.wDepth - 1;
PCXH.wXResolution   = 0;
PCXH.wYResolution   = 0;
PCXH.byReserved     = 0;
PCXH.byPlanes       = (BYTE)PCXCVar.wPlanes;
PCXH.wLineBytes     = PCXCVar.wLineBytes;
PCXH.wPaletteType   = (gbGray ? 2D : 1);
PCXH.wScrWidth      = 0;
PCXH.wScrDepth      = 0;
_fmemset(PCXH.byFiller,0,54);
_lwrite(iTempFile,(LPSTR)&PCXH,sizeof(PCXH));
iReturn       = iCreatePCX(iFile,iTempFile,&PCXCVar,wWidthBytes);
if ( PCXCVar.wBits==8 )
{
```

```
        /* Write The PCX 2D56 - Color Palette */
        _lwrite(iTempFile,(LPSTR)byPCX_Pal,769);
    }
    _lclose(iTempFile);
    _lclose(iFile);
    InvalidateRect(ghChildWnd,NULL,TRUE);
    UpdateWindow(ghChildWnd);
    return 0;
}
int iCreatePCX(int iFile,int iTempFile,LPPCXC_VAR lpPCXCVar,WORD wWidthBytes)
{
    HANDLE hSrcBuff;
    HANDLE hImage;
    HANDLE hTemp;
    LPSTR  lpImage;
    LPSTR  lpTemp;
    LPSTR  lpImageBgn;
    LPSTR  lpPlane0;
    LPSTR  lpPlane1;
    LPSTR  lpPlane2;
    LPSTR  lpPlane3;
    WORD   wi;
    WORD   wj;
    BYTE   byTemp;
    BYTE   byPlane0;
    BYTE   byPlane1;
    BYTE   byPlane2;
    BYTE   byPlane3;
    int    iRatio;
    int    iReturn;
    hSrcBuff     = GlobalAlloc(GHND,(DWORD)MAX_BUFF_SIZE);
    lpPCXCVar->lpDataBuff = GlobalLock(hSrcBuff);
    lpPCXCVar->lpEndBuff  = lpPCXCVar->lpDataBuff;
    lpPCXCVar->wByteCnt   = 0;
    hImage       = GlobalAlloc(GHND,(DWORD)wWidthBytes);
    hTemp        = GlobalAlloc(GHND,(DWORD)(wWidthBytes+1));
    lpImage      = GlobalLock(hImage);
    lpTemp       = GlobalLock(hTemp);
    lpImageBgn   = lpImage;
    for(wi=0;wi<lpPCXCVar->wDepth;wi++)
```