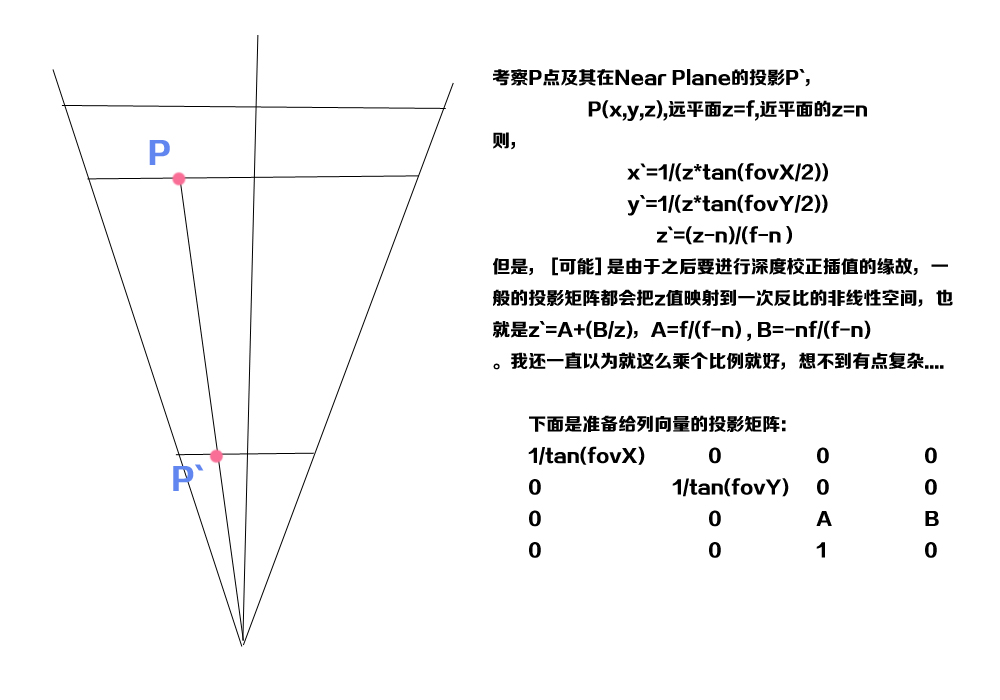
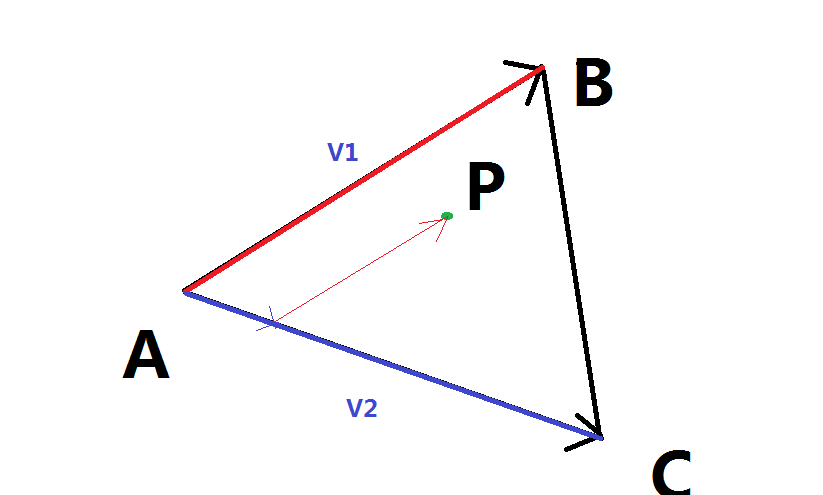
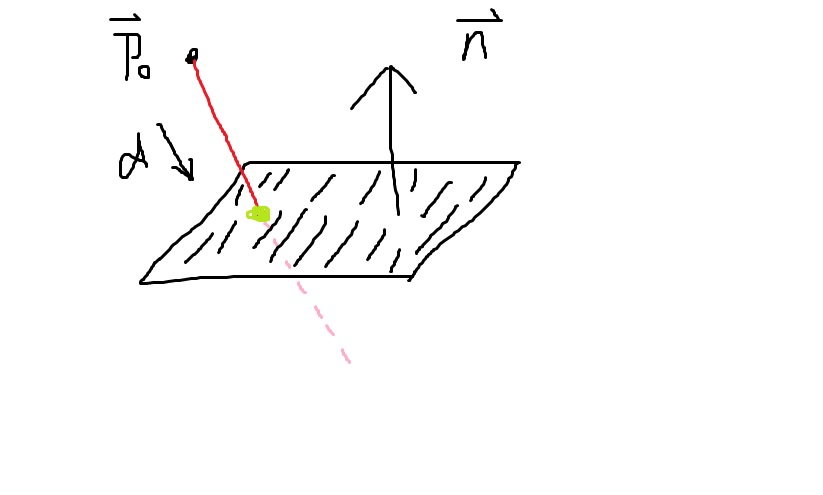
射死大鸡怪词汇表（一些技术细节）

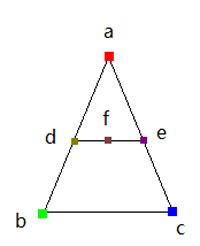
分别投影：

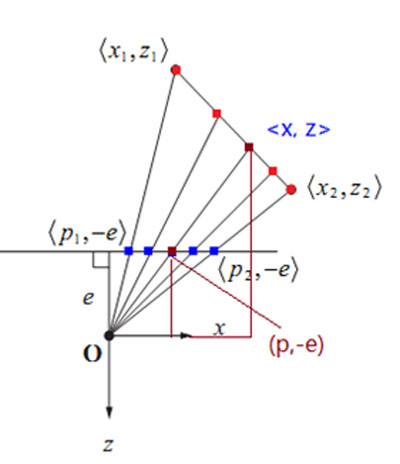
双线性插值：

直线和三角形求交：

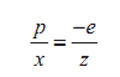


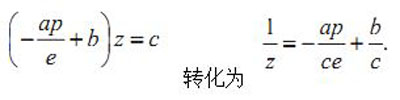
透视校正插值

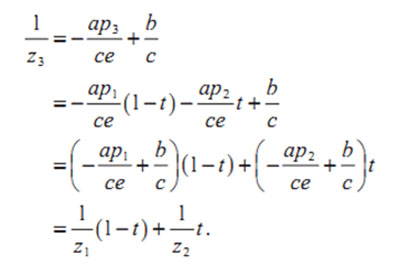
在3D渲染中，输入数据是一些primitive信息，包括顶点位置、颜色、纹理坐标等等。在光栅化阶段，primitive(一般为三角形)被转化成一 系列的fragment(或者称为像素),这些fragment接下来要做ps操作，此时每个fragment都有位置、颜色、纹理坐标这些属性信息，这 些属性信息通过顶点属性用插值方法得到的。比如下面的primtive，输入数据中只有a,b,c三个顶点的信息，则三角形内部经过扫描转化得到的像素f 的颜色则是通过插值得到，d是a、b的差值，e是a、c的差值，然后f又是d、e的插值。

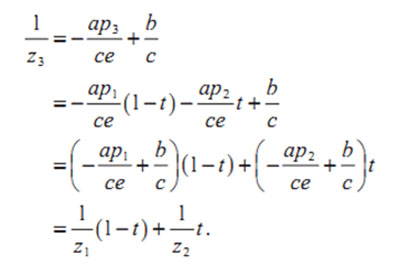
下图是clip裁剪锥体的xz平面，从图中可以看出，在投影平面的均匀插值(一系列蓝色的标记)，它们实际位置距离并不是均匀的，距离投影平面越远，实际位置距离越长，所以这是我们采用线性插值的话，可能最终的结果并不是我们所想要的。

那么如何得到均匀的顶点属性插值呢？稍等一下，我们先看看深度插值：

点O是视点位置，从视点观察物体的投射线和投影平面相交的点即为物体在投影平面上的投射点，比如(x1,z1)在投影平面上的投射点为(p1,-e)，投 射线的方程为ax+bx =c ， 其中c不能等于0，假设点(x,z)的和O在投影平面的交点为(p,-e)(注意z坐标总是-e)，则有

解出x，并把它带入到ax+b=c中，得到

已知端点,的投影坐标为  
,  
，假定p3=(1-t)p1+tp2 (0<=t<=1)，即p3是投影平面上2个端点的线性差值的x分量，则有下面的推导公式：

可见，z的倒数是线性插值，所以我们可以用顶点的z值来插值求得primitive内部fragment的属性值，比如颜色等等。假定的颜色为b1, 的颜色为把b2，则的颜色为b3为：

可见用深度导数来插值顶点属性是合适的。

摘自【GPU技术】透视校正插值 \_ Imagination中文技术社区

[PPM / PGM / PBM 图像文件格式](http://www.cnblogs.com/hnrainll/archive/2011/06/08/2075629.html)

ppm文件是一种图像文件，有其自己的文件格式。ppm文件由两个部分组成：第一个部分是三行ASCII码，这个部分决定了图像的存储格式以及图像的特征；第二个部分就是图像的数据部分，图像就是由这个部分组成的。

　　ppm的第一部分由三行ASCII码组成

第一行是P2/P3/P6  
第二行是图像的大小，先是列像素数，后是行像素数，中间有一个空格  
第三行是一个介于1和65535之间的整数，而且必须是文本的，用来表示每一个像素的一个分量用几个比特表示。

　　三行之后是图像的数据流，从左到右，从上到下。在进行图像数据存储的时候，需要进行数据的格式，假如需要的像素值在0~255之间，那么在进行 数据文件保存的时候，所写入文件的值就必须是以%c的形式输入，而且数据之间没有明显的分离字符，图像处理软件会自动地识别这些像素的值，并给予处理。

PPM->Portable PixMap  
PGM->Portable GreyMap  
PBM->Portable BitMap

PBM支持单色图（1个像素位）  
PGM支持灰度图形，能够读PBM图形和PGM图形，输出PGM图形  
PPM支持真彩色图形，可以读上面所有格式，输出PPM图形

PPM

　　PPM图形文件格式包括两个部分，头部分和图象数据部分。头部分由三部分组成，这三部分由回车或换行分割，但PPM的标准中是要求空格。第一行 通常是P3或P6，说明是PPM格式；第二行是图象的宽度和高度，用ASCII来表示；最后一部分是描述像素的最大颜色组成，这里允许描述超过一个字节 （0-255）的颜色值。另外可以在上面个部分的后面用#来追加注释，注释行是从#到该行末。

　　下面是PPM头的例子：

例子1：  
P6 1024 778 255

例子2：  
P6  
1024 778  
255

例子3：  
P6#PPM文件格式  
1024 778#宽度和高度  
# 注释  
255

　　PPM图象数据的格式依赖于PPM自身的表示，如果是P3格式，数据将以ASCII文本来表示，每个像素的值从0到前面的最大值，每行不应该长于70个字符，如下：

例子4：  
P3  
# example from the man page  
4 4  
15  
0 0 0 0 0 0 0 0 0 15 0 15  
0 0 0 0 15 7 0 0 0 0 0 0  
0 0 0 0 0 0 0 15 7 0 0 0  
15 0 15 0 0 0 0 0 0 0 0 0

　　如果是P6格式，图象数据以字节格式存储，每个色彩成分（R，G，B）一个字节。仅仅在头部的最后一个字段的前面才能有注释，在头部的最后一个字段后面通常是一个回车或换行。P6图象文件比P3文件小，读起来更快。注意，P6文件仅仅用作但字节彩色。

　　但并没有按照格式规约的要求来，通常的习惯，图象从上到下，从左到右被存储。每个像素以一个字节来存储，0表示黑色，255表示白色。色彩成分按照通常的红-绿-蓝顺序爱存储。

PGM

　　该格式文件存储灰度图形，也就是这里每个像素使用一个值来表示而不是3个（R，G，B）。同PPM唯一不同的是头部用P2和P5，分别表示用ASCII和字节码来表示数据。

例如：  
P2  
24 7  
15  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 0  
0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 15 0  
0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11 0 0 0 15 15 15 15 0  
0 3 0 0 0 0 0 7 0 0 0 0 0 11 0 0 0 0 0 15 0 0 0 0  
0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

PBM

　　使用ASCII的0或1方式来表示数据，0表示白色，1表示黑色。与PPM、PGM不同的头部是少了第三行，因为第三行的最大色彩值在这个模式下已经没有意义了；如下：

P1  
# PBM example  
24 7  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1 1 0  
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0  
0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 0  
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0  
0 1 0 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0  
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

PPM文件格式分三种：

1. PPM灰度文件  
　　文件头由3行文本组成，可由fgets读出  
　　1）第一行为“P2"，表示文件类型  
　　2）第二行为图像的宽度和高度  
　　3）第三行为最大的象素值255  
　　接下来是图像数据块。按行顺序存储。每个象素占4个字节，灰度通道为4字节ASCII码表示的整数，高字节在前。左上角为坐标原点。

2. 16位PPM文件（至少适用于读取由DCRAW生成的PPM文件）  
　　文件头由3行文本组成，可由fgets读出  
　　1）第一行为“P6"，表示文件类型  
　　2）第二行为图像的宽度和高度  
　　3）第三行为最大的象素值  
　　接下来是图像数据块。按行顺序存储。每个象素占3个字节，依次为红绿蓝通道，每个通道为1字节整数。左上角为坐标原点。

3. PPM彩色文件  
　　文件头由3行文本组成，可由fgets读出  
　　1）第一行为“P3"，表示文件类型  
　　2）第二行为图像的宽度和高度  
　　3）第三行为最大的象素值255  
　　接下来是图像数据块。按行顺序存储。每个象素占12个字节，依次为红绿蓝通道，每个通道为4字节ASCII码表示的整数，高字节在前。左上角为坐标原点。

　　可移植像素图格式(PPM)，可移植灰度图格式(PGM)和可移植位图格式(PBM)是便于跨平台的图像格式。有时候也被统称为PNM格式。

历史

　　PBM格式由Jef Poskanzer在20世纪80年代发明，为了便于通过电子邮件，用ASCII码表示单色位图，能够承受一般的文本格式的变动。

　　第一个处理PBM格式的工具库是Pbmplus。它由这个格式的发明人Jef Poskanzer开发，在1988年发布。主要包含Jef编写的将PBM转化为已存在的其他图像格式的工具。在1988年末，Jef开发出PGM、 PPM格式以及相关工具，并加入Pbmplus中。Pbmplus的最终发布日期是1991年12月10日。

　　在1993年，Netpbm库开始开发，用来替代不再维护的Pbmplus。它是Pbmplus的简单的重新包装，附加全世界开发者提供的额外功能和修订，可能是目前用的最普遍的处理PBM、PGM和PPM格式的工具库。

文件格式描述

　　这三种格式在颜色的表示上有差异。PBM是单色，PGM是灰度图，PPM使用RGB颜色。

　　每个文件的开头两个字节（ASCII码）作为文件描述子，指出具体格式和编码形式。具体见下表：

|  |  |  |
| --- | --- | --- |
| **文件描述子** | **类型** | **编码** |
| P1 | 位图 | ASCII |
| P2 | 灰度图 | ASCII |
| P3 | 像素图 | ASCII |
| P4 | 位图 | 二进制 |
| P5 | 灰度图 | 二进制 |
| P6 | 像素图 | 二进制 |

摘自PPM \_ PGM \_ PBM 图像文件格式 - Leo Chin - 博客园

Reading and Writing Blocks of Characters and Attributes

The [**ReadConsoleOutput**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684965%28v=vs.85%29.aspx) function copies a rectangular block of character and color attribute data from a console screen buffer into a destination buffer. The function treats the destination buffer as a two-dimensional array of [**CHAR\_INFO**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682013%28v=vs.85%29.aspx) structures. Similarly, the [**WriteConsoleOutput**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms687404%28v=vs.85%29.aspx) function copies a rectangular block of character and color attribute data from a source buffer to a console screen buffer. For more information about reading from or writing to rectangular blocks of screen buffer cells, see[Input and Output Methods](https://msdn.microsoft.com/en-us/library/windows/desktop/ms683497%28v=vs.85%29.aspx).

The following example uses the [**CreateConsoleScreenBuffer**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682122%28v=vs.85%29.aspx) function to create a new screen buffer. After the[**SetConsoleActiveScreenBuffer**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms686010%28v=vs.85%29.aspx) function makes this the active screen buffer, a block of characters and color attributes is copied from the top two rows of the STDOUT screen buffer into a temporary buffer. The data is then copied from the temporary buffer into the new active screen buffer. When the application is finished using the new screen buffer, it calls**SetConsoleActiveScreenBuffer** to restore the original STDOUT screen buffer.

C++

#include <windows.h>

#include <stdio.h>

int main(void)

{

HANDLE hStdout, hNewScreenBuffer;

SMALL\_RECT srctReadRect;

SMALL\_RECT srctWriteRect;

CHAR\_INFO chiBuffer[160]; // [2][80];

COORD coordBufSize;

COORD coordBufCoord;

BOOL fSuccess;

// Get a handle to the STDOUT screen buffer to copy from and

// create a new screen buffer to copy to.

hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);

hNewScreenBuffer = CreateConsoleScreenBuffer(

GENERIC\_READ | // read/write access

GENERIC\_WRITE,

FILE\_SHARE\_READ |

FILE\_SHARE\_WRITE, // shared

NULL, // default security attributes

CONSOLE\_TEXTMODE\_BUFFER, // must be TEXTMODE

NULL); // reserved; must be NULL

if (hStdout == INVALID\_HANDLE\_VALUE ||

hNewScreenBuffer == INVALID\_HANDLE\_VALUE)

{

printf("CreateConsoleScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

// Make the new screen buffer the active screen buffer.

if (! SetConsoleActiveScreenBuffer(hNewScreenBuffer) )

{

printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

// Set the source rectangle.

srctReadRect.Top = 0; // top left: row 0, col 0

srctReadRect.Left = 0;

srctReadRect.Bottom = 1; // bot. right: row 1, col 79

srctReadRect.Right = 79;

// The temporary buffer size is 2 rows x 80 columns.

coordBufSize.Y = 2;

coordBufSize.X = 80;

// The top left destination cell of the temporary buffer is

// row 0, col 0.

coordBufCoord.X = 0;

coordBufCoord.Y = 0;

// Copy the block from the screen buffer to the temp. buffer.

fSuccess = ReadConsoleOutput(

hStdout, // screen buffer to read from

chiBuffer, // buffer to copy into

coordBufSize, // col-row size of chiBuffer

coordBufCoord, // top left dest. cell in chiBuffer

&srctReadRect); // screen buffer source rectangle

if (! fSuccess)

{

printf("ReadConsoleOutput failed - (%d)\n", GetLastError());

return 1;

}

// Set the destination rectangle.

srctWriteRect.Top = 10; // top lt: row 10, col 0

srctWriteRect.Left = 0;

srctWriteRect.Bottom = 11; // bot. rt: row 11, col 79

srctWriteRect.Right = 79;

// Copy from the temporary buffer to the new screen buffer.

fSuccess = WriteConsoleOutput(

hNewScreenBuffer, // screen buffer to write to

chiBuffer, // buffer to copy from

coordBufSize, // col-row size of chiBuffer

coordBufCoord, // top left src cell in chiBuffer

&srctWriteRect); // dest. screen buffer rectangle

if (! fSuccess)

{

printf("WriteConsoleOutput failed - (%d)\n", GetLastError());

return 1;

}

Sleep(5000);

// Restore the original active screen buffer.

if (! SetConsoleActiveScreenBuffer(hStdout))

{

printf("SetConsoleActiveScreenBuffer failed - (%d)\n", GetLastError());

return 1;

}

return 0;

}

摘自Reading and Writing Blocks of Characters and Attributes (Windows)

# SetConsoleActiveScreenBuffer function

Sets the specified screen buffer to be the currently displayed console screen buffer.

## **Syntax**

C++

BOOL WINAPI SetConsoleActiveScreenBuffer(

\_In\_ HANDLE hConsoleOutput

);

## **Parameters**

hConsoleOutput [in]

A handle to the console screen buffer.

## **Return value**

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [**GetLastError**](https://msdn.microsoft.com/en-us/library/windows/desktop/ms679360%28v=vs.85%29.aspx).

## **Remarks**

A console can have multiple screen buffers. **SetConsoleActiveScreenBuffer** determines which one is displayed. You can write to an inactive screen buffer and then use **SetConsoleActiveScreenBuffer** to display the buffer's contents.

## **Examples**

For an example, see [Reading and Writing Blocks of Characters and Attributes](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685032%28v=vs.85%29.aspx).

## **Requirements**

|  |  |
| --- | --- |
| **Minimum supported client** | Windows 2000 Professional [desktop apps only] |
| **Minimum supported server** | Windows 2000 Server [desktop apps only] |
| **Header** | Wincon.h (include Windows.h) |
| **Library** | Kernel32.lib |
| **DLL** | Kernel32.dll |

摘自SetConsoleActiveScreenBuffer function (Windows)

[图像编程学习笔记1——bmp文件结构处理与显示](http://blog.csdn.net/sun1956/article/details/8648460)

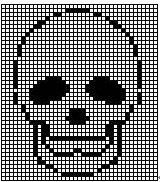
## **文本内容转载自《数字图像处理编程入门》，代码为自己实现**

**1.1图和调色板的概念**

如今Windows(3.x以及95，98，NT)系列已经成为绝大多数用户使用的操作系统，它比DOS成功的一个重要因素是它可视化的漂亮界面。那么Windows是如何显示图象的呢？这就要谈到位图(bitmap)。

我们知道，普通的显示器屏幕是由许许多多点构成的，我们称之为象素。显示时采用扫描的方法：电子枪每次从左到右扫描一行，为每个象素着色，然后从上 到下这样扫描若干行，就扫过了一屏。为了防止闪烁，每秒要重复上述过程几十次。例如我们常说的屏幕分辨率为640×480，刷新频率为70Hz，意思是说 每行要扫描640个象素，一共有480行，每秒重复扫描屏幕70次。

我们称这种显示器为位映象设备。所谓位映象，就是指一个二维的象素矩阵，而位图就是采用位映象方法显示和存储的图象。举个例子，图1.1是一幅普通的黑白位图，图1.2是被放大后的图，图中每个方格代表了一个象素。我们可以看到：整个骷髅就是由这样一些黑点和白点组成的。

**1.1    骷髅       C:\Users\32699\Pictures\1362665246_7595.jpg                                                                     图1.2     放大后的骷髅位图**

那么，彩色图是怎么回事呢？

我们先来说说三元色RGB概念。

我们知道，自然界中的所有颜色都可以由红、绿、蓝(R，G，B)组合而成。有的颜色含有红色成分多一些，如深红；有的含有红色成分少一些，如浅红。 针对含有红色成分的多少，可以分成0到255共256个等级，0级表示不含红色成分；255级表示含有100%的红色成分。同样，绿色和蓝色也被分成 256级。这种分级概念称为量化。

这样，根据红、绿、蓝各种不同的组合我们就能表示出256×256×256，约1600万种颜色。这么多颜色对于我们人眼来说已经足够丰富了。

**表1.1     常见颜色的RGB组合值**

|  |  |  |  |
| --- | --- | --- | --- |
| 颜色 | R | G | B |
| 红 | 255 | 0 | 0 |
| 蓝 | 0 | 255 | 0 |
| 绿 | 0 | 0 | 255 |
| 黄 | 255 | 255 | 0 |
| 紫 | 255 | 0 | 255 |
| 青 | 0 | 255 | 255 |
| 白 | 255 | 255 | 255 |
| 黑 | 0 | 0 | 0 |
| 灰 | 128 | 128 | 128 |

你大概已经明白了，当一幅图中每个象素赋予不同的RGB值时，能呈现出五彩缤纷的颜色了，这样就形成了彩色图。的确是这样的，但实际上的做法还有些差别。

让我们来看看下面的例子。

有一个长宽各为200个象素，颜色数为16色的彩色图，每一个象素都用R、G、B三个分量表示。因为每个分量有256个级别，要用8位(bit)， 即一个字节(byte)来表示，所以每个象素需要用3个字节。整个图象要用200×200×3，约120k字节，可不是一个小数目呀！如果我们用下面的方 法，就能省的多。

因为是一个16色图，也就是说这幅图中最多只有16种颜色，我们可以用一个表：表中的每一行记录一种颜色的R、G、B值。这样当我们表示一个象素的 颜色时，只需要指出该颜色是在第几行，即该颜色在表中的索引值。举个例子，如果表的第0行为255，0，0(红色)，那么当某个象素为红色时，只需要标明 0即可。

让我们再来计算一下：16种状态可以用4位(bit)表示，所以一个象素要用半个字节。整个图象要用200×200×0.5，约20k字节，再加上表占用的字节为3×16=48字节.整个占用的字节数约为前面的1/6，省很多吧？

这张R、G、B的表，就是我们常说的调色板(Palette)，另一种叫法是颜色查找表LUT(Look UpTable)，似乎更确切一些。Windows位图中便用到了调色板技术。其实不光是Windows位图，许多图象文件格式如pcx、tif、gif 等都用到了。所以很好地掌握调色板的概念是十分有用的。

有一种图，它的颜色数高达256×256×256种，也就是说包含我们上述提到的R、G、B颜色表示方法中所有的颜色，这种图叫做真彩色图 (true color)。真彩色图并不是说一幅图包含了所有的颜色，而是说它具有显示所有颜色的能力，即最多可以包含所有的颜色。表示真彩色图时，每个象素直接用 R、G、B三个分量字节表示，而不采用调色板技术。原因很明显：如果用调色板，表示一个象素也要用24位，这是因为每种颜色的索引要用24位(因为总共有 224种颜色，即调色板有224行)，和直接用R，G，B三个分量表示用的字节数一样，不但没有任何便宜，还要加上一个256×256×256×3个字节的大调色板。所以真彩色图直接用R、G、B三个分量表示，它又叫做24位色图。

**1.2****bmp文件格式**

介绍完位图和调色板的概念，下面就让我们来看一看Windows的位图文件(.bmp文件)的格式是什么样子的。

bmp文件大体上分成四个部分，如图1.3所示。

|  |
| --- |
| 位图文件头BITMAPFILEHEADER |
| 位图信息头BITMAPINFOHEADER |
| 调色板Palette |
| 实际的位图数据ImageDate |

**图1.3     Windows位图文件结构示意图**

第一部分为位图文件头**BITMAPFILEHEADER**，是一个结构，其定义如下：

typedefstruct tagBITMAPFILEHEADER {

WORD          bfType;

DWORD bfSize;

WORD          bfReserved1;

WORD          bfReserved2;

DWORDbfOffBits;

}BITMAPFILEHEADER;

这个结构的长度是固定的，为14个字节(WORD为无符号16位整数，DWORD为无符号32位整数)，各个域的说明如下：

**bfType**

指定文件类型，必须是0x424D，即字符串“BM”，也就是说所有.bmp文件的头两个字节都是“BM”。

**bfSize**

指定文件大小，包括这14个字节。

**bfReserved1，bfReserved2**

为保留字，不用考虑

**bfOffBits**

为从文件头到实际的位图数据的偏移字节数，即图1.3中前三个部分的长度之和。

第二部分为位图信息头**BITMAPINFOHEADER**，也是一个结构，其定义如下：

typedefstruct tagBITMAPINFOHEADER{

DWORD biSize;

LONG           biWidth;

LONG           biHeight;

WORD          biPlanes;

WORD          biBitCount

DWORD biCompression;

DWORD biSizeImage;

LONG           biXPelsPerMeter;

LONG           biYPelsPerMeter;

DWORD biClrUsed;

DWORD biClrImportant;

}BITMAPINFOHEADER;

这个结构的长度是固定的，为40个字节(LONG为32位整数)，各个域的说明如下：

**biSize**

指定这个结构的长度，为40。

**biWidth**

指定图象的宽度，单位是象素。

**biHeight**

指定图象的高度，单位是象素。

**biPlanes**

必须是1，不用考虑。

**biBitCount**

指定表示颜色时要用到的位数，常用的值为1(黑白二色图), 4(16色图), 8(256色), 24(真彩色图)(新的.bmp格式支持32位色，这里就不做讨论了)。

**biCompression**

指定位图是否压缩，有效的值为BI\_RGB，BI\_RLE8，BI\_RLE4，BI\_BITFIELDS(都是一些Windows定义好的常量)。 要说明的是，Windows位图可以采用RLE4，和RLE8的压缩格式，但用的不多。我们今后所讨论的只有第一种不压缩的情况，即 biCompression为BI\_RGB的情况。

**biSizeImage**

指定实际的位图数据占用的字节数，其实也可以从以下的公式中计算出来：

biSizeImage=biWidth’ × biHeight

要注意的是：上述公式中的biWidth’必须是4的整倍数(所以不是biWidth，而是biWidth’，表示大于或等于biWidth的，最 接近4的整倍数。举个例子，如果biWidth=240，则biWidth’=240；如果biWidth=241，biWidth’=244)。

如果biCompression为BI\_RGB，则该项可能为零

**biXPelsPerMeter**

指定目标设备的水平分辨率，单位是每米的象素个数，关于分辨率的概念，我们将在第4章详细介绍。

**biYPelsPerMeter**

指定目标设备的垂直分辨率，单位同上。

**biClrUsed**

指定本图象实际用到的颜色数，如果该值为零，则用到的颜色数为2biBitCount。

**biClrImportant**

指定本图象中重要的颜色数，如果该值为零，则认为所有的颜色都是重要的。

第三部分为调色板**Palette**，当然，这里是对那些需要调色板的位图文件而言的。有些位图，如真彩色图，前面已经讲过，是不需要调色板的，BITMAPINFOHEADER后直接是位图数据。

调色板实际上是一个数组，共有biClrUsed个元素(如果该值为零，则有2biBitCount个元素)。数组中每个元素的类型是一个RGBQUAD结构，占4个字节，其定义如下：

typedefstruct tagRGBQUAD {

BYTE   rgbBlue; //该颜色的蓝色分量

BYTE   rgbGreen; //该颜色的绿色分量

BYTE   rgbRed; //该颜色的红色分量

BYTE   rgbReserved; //保留值

} RGBQUAD;

第四部分就是实际的图象数据了。对于用到调色板的位图，图象数据就是该象素颜在调色板中的索引值。对于真彩色图，图象数据就是实际的R、G、B值。下面针对2色、16色、256色位图和真彩色位图分别介绍。

对于2色位图，用1位就可以表示该象素的颜色(一般0表示黑，1表示白)，所以一个字节可以表示8个象素。

对于16色位图，用4位可以表示一个象素的颜色，所以一个字节可以表示2个象素。

对于256色位图，一个字节刚好可以表示1个象素。

对于真彩色图，三个字节才能表示1个象素，哇，好费空间呀！没办法，谁叫你想让图的颜色显得更亮丽呢，有得必有失嘛。

要注意两点：

(1)    每一行的字节数必须是4的整倍数，如果不是，则需要补齐。这在前面介绍biSizeImage时已经提到了。

(2)    一般来说，.bMP文件的数据从下到上，从左到右的。也就是说，从文件中最先读到的是图象最下面一行的左边第一个象素，然后是左边第二个象素……接下来是 倒数第二行左边第一个象素，左边第二个象素……依次类推，最后得到的是最上面一行的最右一个象素。

开发工具：vc++6.0，Win32 控制台程序

**[cpp]** [view plain](file:///C:\Users\32699\Documents\RUANXU\Shoot-The-Chicken-3D-HD-master\doc\%E7%BD%91%E9%A1%B5\%E5%9B%BE%E5%83%8F%E7%BC%96%E7%A8%8B%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B01%E2%80%94%E2%80%94bmp%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84%E5%A4%84%E7%90%86%E4%B8%8E%E6%98%BE%E7%A4%BA%20-%20Sun1956%20-%20%E5%8D%9A%E5%AE%A2%E9%A2%91%E9%81%93%20-%20CSDN.NET.htm) [copy](file:///C:\Users\32699\Documents\RUANXU\Shoot-The-Chicken-3D-HD-master\doc\%E7%BD%91%E9%A1%B5\%E5%9B%BE%E5%83%8F%E7%BC%96%E7%A8%8B%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B01%E2%80%94%E2%80%94bmp%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84%E5%A4%84%E7%90%86%E4%B8%8E%E6%98%BE%E7%A4%BA%20-%20Sun1956%20-%20%E5%8D%9A%E5%AE%A2%E9%A2%91%E9%81%93%20-%20CSDN.NET.htm)

1. /\*\*
2. \* 程序名: WorkBmp.cpp
3. \* 功  能: 读取和显示24位BMP图像,并把图像数据输入到ImageData.txt中
4. \* 24位bmp可以通过画图程序中的另存为的文件类型中可以选择
5. \* bmp文件放到工程目录下
6. \*/
7. #include <iostream.h>
8. #include <stdio.h>
9. #include <windows.h>
10. #include <fstream.h>
11. **int** biWidth;  //图像宽
12. **int** biHeight;  //图像高
13. **int** biBitCount; //图像类型，每像素位数
14. //RGBQUAD \*pColorTable;  //颜色表指针
15. unsigned **char** \*pBmpBuf;  //存储图像数据
16. **int** lineByte;         //图像数据每行字节数
17. /\*\*
18. \* 函数名: readBmp
19. \* 参  数: bmpName -- bmp文件名
20. \* 功  能: 读入bmp文件，并获取相应的信息
21. \*/
22. **bool** readBmp(**char** \*bmpName)
23. {
24. **FILE** \*fp;
25. **if**( (fp = fopen(bmpName,"rb")) == NULL)  //以二进制的方式打开文件
26. {
27. cout<<"The file "<<bmpName<<"was not opened"<<endl;
28. **return** FALSE;
29. }
30. **if**(fseek(fp,**sizeof**(BITMAPFILEHEADER),SEEK\_CUR))  //跳过BITMAPFILEHEADE
31. {
32. cout<<"跳转失败"<<endl;
33. **return** FALSE;
34. }
35. BITMAPINFOHEADER infoHead;
36. fread(&infoHead,**sizeof**(BITMAPINFOHEADER),1,fp);   //从fp中读取BITMAPINFOHEADER信息到infoHead中,同时fp的指针移动
37. biWidth = infoHead.biWidth;
38. biHeight = infoHead.biHeight;
39. biBitCount = infoHead.biBitCount;
40. lineByte = (biWidth\*biBitCount/8+3)/4\*4;   //lineByte必须为4的倍数
41. //24位bmp没有颜色表，所以就直接到了实际的位图数据的起始位置
42. pBmpBuf = **new** unsigned **char**[lineByte \* biHeight];
43. fread(pBmpBuf,**sizeof**(**char**),lineByte \* biHeight,fp);
44. fclose(fp);   //关闭文件
45. **return** TRUE;
47. }
48. /\*\*
49. \* 函数名: saveBmp
50. \* 参  数: bmpName -- bmp文件名
51. \* 功  能: 将bmp位图文件的相关信息，写入新创建的文件中
52. \*/
53. **bool** saveBmp(**char** \*bmpName)
54. {
55. **FILE** \*fp;
56. **if**( (fp = fopen(bmpName,"wb") )== NULL)   //以二进制写入方式打开
57. {
58. cout<<"打开失败!"<<endl;
59. **return** FALSE;
60. }
61. //设置BITMAPFILEHEADER参数
62. BITMAPFILEHEADER fileHead;
63. fileHead.bfType = 0x4D42;
64. fileHead.bfSize = **sizeof**(BITMAPFILEHEADER) + **sizeof**(BITMAPINFOHEADER) + lineByte \* biHeight;
65. fileHead.bfReserved1 = 0;
66. fileHead.bfReserved2 = 0;
67. fileHead.bfOffBits = **sizeof**(BITMAPFILEHEADER) + **sizeof**(BITMAPINFOHEADER);
68. fwrite(&fileHead,**sizeof**(BITMAPFILEHEADER),1,fp);
69. //设置BITMAPINFOHEADER参数
70. BITMAPINFOHEADER infoHead;
71. infoHead.biSize = 40;
72. infoHead.biWidth = biWidth;
73. infoHead.biHeight = biHeight;
74. infoHead.biPlanes = 1;
75. infoHead.biBitCount = biBitCount;
76. infoHead.biCompression = BI\_RGB;
77. infoHead.biSizeImage = lineByte \* biHeight;
78. infoHead.biXPelsPerMeter = 0;
79. infoHead.biYPelsPerMeter = 0;
80. infoHead.biClrUsed = 0;
81. infoHead.biClrImportant = 0;
82. //写入
83. fwrite(&infoHead,**sizeof**(BITMAPINFOHEADER),1,fp);
84. fwrite(pBmpBuf,**sizeof**(**char**),lineByte \* biHeight,fp);
85. fclose(fp);    //关闭文件
86. **return** TRUE;

89. }
90. /\*\*
91. \* 函数名: work
92. \* 功  能: 处理位图信息，并将位图数据保存到ImageData文件中
93. \*/
94. **void** work()
95. {
96. **char** readFileName[] = "nv.BMP";   //定义要读入的文件名
97. **if**(FALSE == readBmp(readFileName))
98. cout<<"readfile error!"<<endl;
99. //输出图像的信息
100. cout<<"Width = "<<biWidth<<" Height = "<<biHeight<<" biBitCount="<<biBitCount<<endl;
101. ofstream outfile("ImageData.txt",ios::in | ios::trunc);
102. **if**(!outfile)
103. {
104. cout<<"open error"<<endl;
105. **return** ;
106. }
107. **int** count = 0;
108. //图像数据信息是从左下角按行开始存储的
109. **for**(**int** i = 0; i < biHeight; i++ )
110. {
111. **for**(**int** j = 0; j < biWidth; j++ )
112. {
113. **for**(**int** k = 0; k < 3; k++ )
114. {
115. **int** temp = \*(pBmpBuf + i \* lineByte + j + k);
116. count++;
117. outfile<<temp<<" ";
118. **if**(count % 8 == 0)
119. {
120. outfile<<endl;
121. }
122. }
123. }
124. }
125. cout<<"总的像素数:"<<count / 3<<endl;
127. **char** writeBmpName[] = "nvcpy.BMP";
128. saveBmp(writeBmpName);
129. **delete** []pBmpBuf;  //释放内存
130. }
132. **int** main()
133. {
134. work();
135. **return** 0;
136. }

**[cpp]** [view plain](file:///C:\Users\32699\Documents\RUANXU\Shoot-The-Chicken-3D-HD-master\doc\%E7%BD%91%E9%A1%B5\%E5%9B%BE%E5%83%8F%E7%BC%96%E7%A8%8B%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B01%E2%80%94%E2%80%94bmp%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84%E5%A4%84%E7%90%86%E4%B8%8E%E6%98%BE%E7%A4%BA%20-%20Sun1956%20-%20%E5%8D%9A%E5%AE%A2%E9%A2%91%E9%81%93%20-%20CSDN.NET.htm) [copy](file:///C:\Users\32699\Documents\RUANXU\Shoot-The-Chicken-3D-HD-master\doc\%E7%BD%91%E9%A1%B5\%E5%9B%BE%E5%83%8F%E7%BC%96%E7%A8%8B%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B01%E2%80%94%E2%80%94bmp%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84%E5%A4%84%E7%90%86%E4%B8%8E%E6%98%BE%E7%A4%BA%20-%20Sun1956%20-%20%E5%8D%9A%E5%AE%A2%E9%A2%91%E9%81%93%20-%20CSDN.NET.htm)

1. /\*\*
2. \* 程序名: WorkBmp.cpp
3. \* 功  能: 读取和显示24位BMP图像,并把图像数据输入到ImageData.txt中
4. \* 24位bmp可以通过画图程序中的另存为的文件类型中可以选择
5. \* bmp文件放到工程目录下
6. \*/
7. #include <iostream.h>
8. #include <stdio.h>
9. #include <windows.h>
10. #include <fstream.h>
11. **int** biWidth;  //图像宽
12. **int** biHeight;  //图像高
13. **int** biBitCount; //图像类型，每像素位数
14. //RGBQUAD \*pColorTable;  //颜色表指针
15. unsigned **char** \*pBmpBuf;  //存储图像数据
16. **int** lineByte;         //图像数据每行字节数
17. /\*\*
18. \* 函数名: readBmp
19. \* 参  数: bmpName -- bmp文件名
20. \* 功  能: 读入bmp文件，并获取相应的信息
21. \*/
22. **bool** readBmp(**char** \*bmpName)
23. {
24. **FILE** \*fp;
25. **if**( (fp = fopen(bmpName,"rb")) == NULL)  //以二进制的方式打开文件
26. {
27. cout<<"The file "<<bmpName<<"was not opened"<<endl;
28. **return** FALSE;
29. }
30. **if**(fseek(fp,**sizeof**(BITMAPFILEHEADER),SEEK\_CUR))  //跳过BITMAPFILEHEADE
31. {
32. cout<<"跳转失败"<<endl;
33. **return** FALSE;
34. }
35. BITMAPINFOHEADER infoHead;
36. fread(&infoHead,**sizeof**(BITMAPINFOHEADER),1,fp);   //从fp中读取BITMAPINFOHEADER信息到infoHead中,同时fp的指针移动
37. biWidth = infoHead.biWidth;
38. biHeight = infoHead.biHeight;
39. biBitCount = infoHead.biBitCount;
40. lineByte = (biWidth\*biBitCount/8+3)/4\*4;   //lineByte必须为4的倍数
41. //24位bmp没有颜色表，所以就直接到了实际的位图数据的起始位置
42. pBmpBuf = **new** unsigned **char**[lineByte \* biHeight];
43. fread(pBmpBuf,**sizeof**(**char**),lineByte \* biHeight,fp);
44. fclose(fp);   //关闭文件
45. **return** TRUE;
47. }
48. /\*\*
49. \* 函数名: saveBmp
50. \* 参  数: bmpName -- bmp文件名
51. \* 功  能: 将bmp位图文件的相关信息，写入新创建的文件中
52. \*/
53. **bool** saveBmp(**char** \*bmpName)
54. {
55. **FILE** \*fp;
56. **if**( (fp = fopen(bmpName,"wb") )== NULL)   //以二进制写入方式打开
57. {
58. cout<<"打开失败!"<<endl;
59. **return** FALSE;
60. }
61. //设置BITMAPFILEHEADER参数
62. BITMAPFILEHEADER fileHead;
63. fileHead.bfType = 0x4D42;
64. fileHead.bfSize = **sizeof**(BITMAPFILEHEADER) + **sizeof**(BITMAPINFOHEADER) + lineByte \* biHeight;
65. fileHead.bfReserved1 = 0;
66. fileHead.bfReserved2 = 0;
67. fileHead.bfOffBits = **sizeof**(BITMAPFILEHEADER) + **sizeof**(BITMAPINFOHEADER);
68. fwrite(&fileHead,**sizeof**(BITMAPFILEHEADER),1,fp);
69. //设置BITMAPINFOHEADER参数
70. BITMAPINFOHEADER infoHead;
71. infoHead.biSize = 40;
72. infoHead.biWidth = biWidth;
73. infoHead.biHeight = biHeight;
74. infoHead.biPlanes = 1;
75. infoHead.biBitCount = biBitCount;
76. infoHead.biCompression = BI\_RGB;
77. infoHead.biSizeImage = lineByte \* biHeight;
78. infoHead.biXPelsPerMeter = 0;
79. infoHead.biYPelsPerMeter = 0;
80. infoHead.biClrUsed = 0;
81. infoHead.biClrImportant = 0;
82. //写入
83. fwrite(&infoHead,**sizeof**(BITMAPINFOHEADER),1,fp);
84. fwrite(pBmpBuf,**sizeof**(**char**),lineByte \* biHeight,fp);
85. fclose(fp);    //关闭文件
86. **return** TRUE;

89. }
90. /\*\*
91. \* 函数名: work
92. \* 功  能: 处理位图信息，并将位图数据保存到ImageData文件中
93. \*/
94. **void** work()
95. {
96. **char** readFileName[] = "nv.BMP";   //定义要读入的文件名
97. **if**(FALSE == readBmp(readFileName))
98. cout<<"readfile error!"<<endl;
99. //输出图像的信息
100. cout<<"Width = "<<biWidth<<" Height = "<<biHeight<<" biBitCount="<<biBitCount<<endl;
101. ofstream outfile("ImageData.txt",ios::in | ios::trunc);
102. **if**(!outfile)
103. {
104. cout<<"open error"<<endl;
105. **return** ;
106. }
107. **int** count = 0;
108. //图像数据信息是从左下角按行开始存储的
109. **for**(**int** i = 0; i < biHeight; i++ )
110. {
111. **for**(**int** j = 0; j < biWidth; j++ )
112. {
113. **for**(**int** k = 0; k < 3; k++ )
114. {
115. **int** temp = \*(pBmpBuf + i \* lineByte + j + k);
116. count++;
117. outfile<<temp<<" ";
118. **if**(count % 8 == 0)
119. {
120. outfile<<endl;
121. }
122. }
123. }
124. }
125. cout<<"总的像素数:"<<count / 3<<endl;
127. **char** writeBmpName[] = "nvcpy.BMP";
128. saveBmp(writeBmpName);
129. **delete** []pBmpBuf;  //释放内存
130. }
132. **int** main()
133. {
134. work();
135. **return** 0;
136. }