User manual for old ppm functions

libppm

Updated: 22 July 2004
Table Of Contents

NAME

libppm - functions for PPM programs

SYNOPSIS

#include <ppm.h>

void ppm_init(int * argcP, char * argv[]);

pixel ** ppm_allocarray(int cols,int rows);

pixel * ppm_allocrow(int cols);

void ppm_freearray(pixel ** pixels, int rows);

void ppm_freerow(pixel * pixelrow);

void ppm_readppminit(FILE * fp, int * colsP, int * rowsP, pixval * maxvalP,int * formatP);

void ppm_readppmrow(FILE *fp, pixel * pixelrow, int cols, pixval maxval, int format);

pixel ** ppm_readppm(FILE * fp, int * colsP, int * rowsP, pixvalP * maxvalP);

void ppm_writeppminit(FILE * fp, int cols, int rows, pixval maxval, int forceplain);

void ppm_writeppmrow(FILE * fp, pixel * pixelrow, int cols, pixval maxval, int forceplain);

void ppm_writeppm(FILE * fp, pixel ** pixels, int cols, int rows, pixval maxval, int forceplain);

void ppm_writeppm(FILE * fp, pixel ** pixels, int cols, int rows, pixval maxval, int forceplain);

void ppm_nextimage(FILE * file, int * const eofP);

void ppm_check(FILE * file, const enum pm_check_type check_type, const int format, const int cols,
const int rows, const int maxval,
enum pm_check_code * const retval);

typedef ... pixel; typedef ... pixval;

#define PPM_MAXMAXVAL ...

#define PPM_OVERALLMAXVAL ...

#define PPM_FORMAT ...

#define RPPM_FORMAT ...

#define PPM_TYPE PPM_FORMAT

#define PPM_FORMAT_TYPE(format) ...

extern pixval ppm_pbmmaxval;

pixval PPM_GETR(pixel p)

pixval PPM_GETG(pixel p)

pixval PPM_GETB(pixel p)

void PPM_ASSIGN(pixel p, pixval red, pixval grn, pixval blu)

int PPM_EQUAL(pixel *p*, **pixel** *q*)

int PPM_ISGRAY(pixel *p*)

void PPM_DEPTH(pixel *newp*, **pixel** *p*, **pixval** *oldmaxval*, **pixval** *newmaxval*)

pixel ppm_parsecolor(char * *colorname*, **pixval** *maxval*)

char * ppm_colorname(pixel * *colorP*, **pixval** *maxval*, **int** *hexok*)

void ppm_readcolornamefile(const char **fileName*, **int** *mustOpen*, **colorhash_table *** *chtP*, **const char ***** *colornamesP*)

DESCRIPTION

These library functions are part of Netpbm.

TYPES AND CONSTANTS

Each **pixel** contains three **pixval**s, each of which should contain only the values between **0** and **PPM_MAXMAXVAL**. **ppm_pbmmaxval** is the maxval used when a PPM program reads a PBM file. Normally it is 1; however, for some programs, a larger value gives better results.

MANIPULATING PIXELS

The macros **PPM_GETR**, **PPM_GETG**, and **PPM_GETB** retrieve the red, green, or blue sample, respectively, from the given pixel.

The **PPM_ASSIGN** macro assigns the given values to the red, green, and blue samples of the given pixel.

The **PPM_EQUAL** macro tests two pixels for equality.

The **PPM_ISGRAY** macro tests a pixel for being gray. It returns true if and only if the color of pixel *p* is black, white, or gray.

The **PPM_DEPTH** macro scales the colors of pixel *p* according the old and new maxvals and assigns the new values to *newp*. It is intended to make writing ppmtowhatever easier.

The **PPM_LUMIN**, **PPM_CHROM_R**, and **PPM_CHROM_B** macros determine the luminance, red chrominance, and blue chrominance, respectively, of the pixel *p*. The scale of all these values is the same as the scale of the input samples (i.e. 0 to maxval for luminance, -maxval/2 to maxval/2 for chrominance).

Note that the macros do it by floating point multiplication. If you are computing these values over an entire image, it may be significantly faster to do it with multiplication tables instead. Compute all the possible products once up front, then for each pixel, just look up the products in the tables.

INITIALIZATION

ppm_init() is identical to **pm_init**.

MEMORY MANAGEMENT

ppm_allocarray() allocates an array of pixels.

ppm_allocrow() allocates a row of the given number of pixels.

ppm_freearray() frees the array allocated with **ppm_allocarray()** containing the given number of rows.

ppm_freerow() frees a row of pixelss allocated with **ppm_allocrow**().

READING FILES

If a function in this section is called on a PBM or PGM format file, it translates the PBM or PGM file into a PPM file on the fly and functions as if it were called on the equivalent PPM file. The *format* value returned by **ppm_readppminit**() is, however, not translated. It represents the actual format of the PBM or PGM file.

ppm_readppminit() reads the header of a PPM file, returning all the information from the header and leaving the file positioned just after the header.

ppm_readppmrow() reads a row of pixels into the *pixelrow* array. *format*, *cols*, and *maxval* are the values returned by **ppm_readppminit()**.

ppm_readppm() reads an entire PPM image into memory, returning the allocated array as its return value and returning the information from the header as *rows*, *cols*, and *maxval*. This function combines **ppm_readppminit()**, **ppm_allocarray()**, and **ppm_readppmrow()**.

WRITING FILES

ppm_writeppminit() writes the header for a PPM file and leaves it positioned just after the header.

forceplain is a logical value that tells **ppm_writeppminit()** to write a header for a plain PPM format file, as opposed to a raw PPM format file.

ppm_writeppmrow() writes the row *pixelrow* to a PPM file. For meaningful results, *cols*, *maxval*, and *forceplain* must be the same as was used with **ppm_writeppminit()**.

ppm_writeppm() write the header and all data for a PPM image. This function combines **ppm_writeppminit()** and **ppm_writeppmrow()**.

MISCELLANEOUS

ppm_nextimage() positions a PPM input file to the next image in it (so that a subsequent **ppm_readppminit()** reads its header).

ppm_nextimage() is analogous to **pbm_nextimage()**, but works on PPM, PGM, and PBM files.

ppm_check() checks for the common file integrity error where the file is the wrong size to contain all the image data.

ppm_check() is analogous to **pbm_check()**, but works on PPM, PGM, and PBM files.

COLOR

Luminance, Chrominance (YcbCr)

```
float PPM_LUMIN(pixel p);
float PPM_CHROM_B(pixel p);
float PPM_CHROM_R(pixel p);
```

PPM_LUMIN takes a **pixel** as an argument and returns the luminance of that pixel, with the same maxval as the pixel (e.g. if the pixel's maxval is 255, a **PPM_LUMIN** value of 255 means fully luminant).

PPM_CHROM_B and **PPM_CHROM_R** are similar, for the red and blue chrominance values.

```
pixel
ppm_color_from_ycbcr(unsigned int y,
                     int      cb,
                     int      cr);
```

ppm_color_from_ycbcr() converts in the other direction. Given luminance and chrominance, it returns a pixel value.

Hue, Saturation, Value (HSV)

```
struct hsv {
    double h;  /* hue (degrees)  0..360 */
    double s;  /* saturation (0-1) */
    double v;  /* value (0-1) */
};
```

```
pixel
ppm_color_from_hsv(struct hsv const hsv,
                   pixval     const maxval);

struct hsv
ppm_hsv_from_color(pixel  const color,
                   pixval const maxval);
```

These convert a color between from **pixel** (RGB) form and HSV.

```
pixval
ppm_saturation(pixel  const p,
               pixval const maxval);
```

This gives you the saturation of a color, as a pixval. (e.g. if the saturation of *p* is 50% and *maxval* is 100, **ppm_saturation**() returns 50).

Berlin-Kay Color

Brent Berlin and Paul Kay in 1969 did a study which identified a set of 11 basic colors people universally recognize. They are:

black

gray

white

red

orange

yellow

green

blue

violet

purple

brown

The **bk_color** type represents a color from this set:

```
typedef enum {
    BKCOLOR_BLACK = 0,
    BKCOLOR_GRAY,
    BKCOLOR_WHITE,
    BKCOLOR_RED,
    BKCOLOR_ORANGE,
    BKCOLOR_YELLOW,
    BKCOLOR_GREEN,
    BKCOLOR_BLUE,
    BKCOLOR_VIOLET,
    BKCOLOR_PURPLE,
    BKCOLOR_BROWN
} bk_color;
```

You can use this as an index of an array, in which case you might also want macro **BKCOLOR_COUNT**, which is the number of colors in the set (11).

To translate between the **bk_color** type and the English names of the colors, use **ppm_bk_color_from_name**() and **ppm_name_from_bk_color**():

```
bk_color
ppm_bk_color_from_name(const char * name);

const char *
ppm_name_from_bk_color(bk_color bkColor);
```

ppm_bk_color_from_color() tells you to which Berlin-Kay color a certain color is closest, by way of a fuzzy color matching algorithm:

```
bk_color
ppm_bk_color_from_color(pixel  color,
                        pixval maxval);
```

maxval is the maxval on which *color* is based.

ppm_color_from_bk_color() converts the opposite way: given a Berlin-Kay color, it gives the color, in **pixel** form, that best represents it.

```
pixel
ppm_color_from_bk_color(bk_color bkColor,
                        pixval   maxval);
```

maxval is the maxval on which the returned color is based.

All of the facilities in this section were new in Netpbm 10.34 (June 2006).

COLOR NAMES

System Color Dictionary

Netpbm uses the system's X11 color dictionary (usually in **/usr/lib/X11/rgb.txt**). This is the same file the X Window System typically uses to associate colors with their names.

The color dictionary that Netpbm uses is in the file whose name is the value of the **RGBDEF** environment variable. If **RGBDEF** is not set, Netpbm defaults to the first existing file from this list:

1. **/usr/lib/X11/rgb.txt**
2. **/usr/openwinlib/rgb.txt**
3. **/usr/X11R6/lib/X11/rgb.txt**

You can see the color names from a typical X11 color dictionary, which is probably very close to what is on your system, along with the colors, here. This website shows a bunch of other versions you could use.

Netpbm is packaged with a color dictionary. A standard Netpbm installation installs this file as "misc/rgb.txt" in the Netpbm directory. This color dictionary has colors from everywhere the Netpbm maintainer could find them, and is a superset of XFree 86's color dictionary.

ppm_parsecolor

ppm_parsecolor() interprets a color specification and returns a pixel of the color that it indicates. The color specification is ASCII text, in one of these formats:

a name, as defined in the system color dictionary

An X11-style hexadecimal specifier: `rgb:`*r*`/`*g*`/`*b*, where *r*, *g*, and *b* are each 1- to 4-digit hexadecimal numbers. For each, the maxval is the maximum number that can be represented in the number of hexadecimal digits given. Example: `rgb:01/ff/8000` specifies 1/255 red intensity, maximum green intensity, and about half blue intensity.

An X11-style decimal specifier: `rgbi:`*r*`/`*g*`/`*b*, where *r*, *g*, and *b* are floating point numbers from 0 to 1.

an old-X11-style hexadecimal triple: `#rgb`, `#rrggbb`, `#rrrgggbbb`, or `#rrrrggggbbbb`.

A triplet of decimal floating point numbers from 0.0 to 1.0, representing red, green, and blue intensities respectively, separated by commas. E.g. `1.0,0.5,.25`. This is for backwards compatibility; it was in use before MIT came up with the similar and preferred rgbi style).

If the color specification does not conform to any of these formats, including the case that it is a name, but is not in the system color dictionary, **ppm_parsecolor**() throws an error.

ppm_colorname

ppm_colorname() returns a string that describes the color of the given pixel. If a system color dictionary is available and the color appears in it, **ppm_colorname()** returns the name of the color from the file. If the color does not appear in a system color dictionary and *hexok* is true, **ppm_colorname()** returns a hexadecimal color specification triple (#rrggbb). If a system color dictionary is available but the color does not appear in it and *hexok* is false, **ppm_colorname()** returns the name of the closest matching color in the color file. Finally, if there is no system color dictionary available and *hexok* is false, **ppm_colorname()** fails and exits the program with an error message.

The string returned is in static libppm library storage which is overwritten by every call to **ppm_colorname()**.

ppm_readcolornamefile

ppm_readcolornamefile() reads the entire contents of the color dictionary in the file named *fileName* into data structures you can use to access it easily.

The function returns all the color names as an array of null-terminated strings. It mallocs the space for this array and returns its address at *colornamesP*. **(*colornamesP)[*i*]** is the address of the first character in the null-terminated string that is the name of the *i*th color in the dictionary.

The function also returns a **colorhash_table** (see COLOR INDEXING) that matches all these color names up to the colors they represent. It mallocs the space for the **colorhash_table** and returns its address at *chtP*. The number that the **colorhash_table** associates with each color is the index into the color name array described above of the name of that color.

You may specify a null pointer for *fileName* to indicate the default color dictionary.

mustOpen is a boolean. If it is nonzero, the function fails and aborts the program if it is unable to open the specified color dictionary file. If it is zero, though, it simply treats an unopenable color dictionary as an empty one. The colorhash and color name array it returns contain no colors or names.

ppm_readcolornamefile() was new in Netpbm 10.15 (April 2003).

COLOR INDEXING

Sometimes in processing images, you want to associate a value with a particular color. Most often, that's because you're generating a color mapped graphics format. In a color mapped graphics format, the raster contains small numbers, and the file contains a color map that tells what color each of those small numbers refers to. If your image has only 256 colors, but each color takes 24 bits to describe, this can make your output file much smaller than a straightforward RGB raster would.

So, continuing the above example, say you have a **pixel** value for chartreuse and in your output file and you are going to represent chartreuse by the number 12. You need a data structure that allows your program quickly to find out that the number for a chartreuse **pixel** is 12. Netpbm's color indexing data types and functions give you that.

colorhash_table is a C data type that associates an integer with each of an arbitrary number of colors. It is a hash table, so it uses far less space than an array indexed by the color's RGB values would.

The problem with a **colorhash_table** is that you can only look things up in it. You can't find out what colors are in it. So Netpbm has another data type for representing the same information, the poorly but historically named **colorhist_vector**. A **colorhist_vector** is just an array. Each entry represents a color and contains the color's value (as a **pixel**) and the integer value associated with it. The entries are filled in starting with subscript 0 and going consecutively up for the number of colors in the histogram.

(The reason the name is poor is because a color histogram is only one of many things that could be represented by it).

colorhash_table ppm_alloccolorhash()

This creates a **colorhash_table** using dynamically allocated storage. There are no colors in it. If there is not enough storage, it exits the program with an error message.

void ppm_freecolorhash()

This destroys a **ppm_freecolorhash** and frees all the storage associated with it.

int ppm_addtocolorhash(colorhash_table cht, const pixel * const colorP, const int value)

This adds the specified color to the specified **colorhash_table**
and associates the specified value with it.

You must ensure that the color you are adding isn't already present in the **colorhash_table**.

There is no way to update an entry or delete an entry from a **colorhash_table**.

int ppm_lookupcolor(const colorhash_table cht, const pixel * const colorP)

This looks up the specified color in the specified **colorhash_table**. It returns the integer value associated with that color.

If the specified color is not in the hash table, the function returns -1. (So if you assign the value -1 to a color, the return value is ambiguous).

colorhist_vector ppm_colorhashtocolorhist(const colorhash_table cht, const int ncolors)

This converts a **colorhash_table** to a **colorhist_vector**. The return value is a new **colorhist_vector** which you must eventually free with **ppm_freecolorhist()**.

ncolors is the number of colors in **cht**. If it has more colors than that, **ppm_colorhashtocolorhist** does not create a **colorhist_vector** and returns NULL.

colorhash_table ppm_colorhisttocolorhash(const colorhist_vector chv, const int ncolors)

This poorly named function does not convert from a **colorhist_vector** to a **colorhash_table**.

It does create a **colorhash_table** based on a **colorhist_vector** input, but the integer value for a given color in the output is not the same as the integer value for that same color in the input. **ppm_colorhisttocolorhash()** ignores the integer values in the input. In the output, the integer value for a color is the index in the input **colorhist_vector** for that color.

You can easily create a color map for an image by running **ppm_computecolorhist()** over the image, then **ppm_colorhisttocolorhash()** over the result. Now you can use **ppm_lookupcolor()** to find a unique color index for any pixel in the input.

If the same color appears twice in the input, **ppm_colorhisttocolorhash()** exit the program with an error message.

ncolors is the number of colors in **chv**.

The return value is a new **colorhash_table** which you must eventually free with **ppm_freecolorhash()**.

COLOR HISTOGRAMS

The Netpbm libraries give you functions to examine a Netpbm image and determine what colors are in it and how many pixels of each color are in it. This information is known as a color histogram. Netpbm uses its **colorhash_table** data type to represent a color histogram.

colorhash_table ppm_computecolorhash(pixel ** const pixels, const int cols, const int rows, const int maxcolors, int* const colorsP)

This poorly but historically named function generates a **colorhash_table** whose value for each color is the number of pixels in a specified image that have that color. (I.e. a color histogram). As a bonus, it returns the number of colors in the image.

(It's poorly named because not all **colorhash_table**s are color histograms, but that's all it generates).

pixels, **cols**, and **rows** describe the input image.

maxcolors is the maximum number of colors you want processed. If there are more colors that that in the input image, **ppm_computecolorhash()** returns NULL as its return value and stops processing as soon as it discovers this. This makes it run faster and use less memory. One use for **maxcolors** is when you just want to find out whether or not the image has more than N colors and don't want to wait to generate a huge color table if so. If you don't want any limit on the number of colors, specify **maxcolors=0**.

ppm_computecolorhash() returns the actual number of colors in the image as ***colorsP**, but only if it is less than or equal to **maxcolors**.

colorhash_table ppm_computecolorhash2(FILE * const ifp, const int cols, const int rows, const pixval maxval, const int format, const int maxcolors, int* const colorsP)

This is the same as **ppm_computecolorhash()** except that instead of feeding it an array of pixels in storage, you give it an open file stream and it reads the image from the file. The file must be positioned after the header, at the raster. Upon return, the file is still open, but its position is undefined.

maxval and **format** are the values for the image (i.e. information from the file's header).

colorhist_vector ppm_computecolorhist(pixel ** pixels, int cols, int rows, int maxcolors, int * colorsP)

This is like **ppm_computecolorhash**() except that it creates a **colorhist_vector** instead of a **colorhash_table**.

If you supply a nonzero **maxcolors** argument, that is the maximum number of colors you expect to find in the input image. If there are more colors than you say in the image, **ppm_computecolorhist()** returns a null pointer as its return value and nothing meaningful as ***colorsP**.

If not, the function returns the new **colorhist_vector** as its return value and the actual number of colors in the image as ***colorsP**. The returned array has space allocated for the specified number of colors regardless of how many actually exist. The extra space is at the high end of the array and is available for your use in expanding the **colorhist_vector**.

If you specify **maxcolors=0**, there is no limit on the number of colors returned and the return array has space for 5 extra colors at the high end for your use in expanding the **colorhist_vector**.

colorhist_vector ppm_computecolorhist2(FILE * ifp, int cols, int rows, int maxcolors, pixval maxval, int format, int * colorsP)

This is the same as **ppm_computecolorhist()** except that instead of feeding it an array of pixels in storage, you give it an open file stream and it reads the image from the file. The file must be positioned after the header, at the raster. Upon return, the file is still open, but its position is undefined.

SEE ALSO

pbm, **pgm**, **libpbm**

AUTHOR

Table Of Contents