An Introduction to Video Compression in C/C++

Fore June

Chapter 1

Macroblocks

5.1 Introduction

A PC image or a frame with moderate size usually consists of many pixels and requires a large amount of storage space and computing power to process it. For example, an image of size 240 x 240 has 57600 pixels and requires $\frac{3}{2} \times 57600 = 86,400$ bytes of storage space if **4:2:0** format is used. It is difficult and inconvenient to process all of these data simultaneously. In order to make things more manageable, an image is decomposed into **macroblocks**. A macroblock is a 16 x 16 pixel-region, which is the basic unit for processing a frame and is used in video compression standards like MPEG, H.261, H.263, and H.264. A macroblock has a total of $16 \times 16 = 256$ pixels.

In our coding, we shall also process an image in the units of macroblocks. For simplicity and the convenience of discussion, we assume that each of our frames consists of an integral number of macroblocks. That is, both the width and height of an image are divisible by 16. The Common Intermediate Format (CIF) discussed in Chapter 4 also has a resolution of 352×288 that corresponds to 22×18 macroblocks. In addition, we shall use the **4:2:0** YCbCr format; a macroblock then consists of a 16×16 Y sample block, an 8×8 Cb sample block and an 8×8 Cr sample block. To better organize the data, we further divide the 16×16 Y sample block into four 8×8 sample blocks. Therefore, a **4:2:0** macroblock has a total of six 8×8 sample blocks; we label these blocks from 0 to 5 as shown in Figure 5-1:

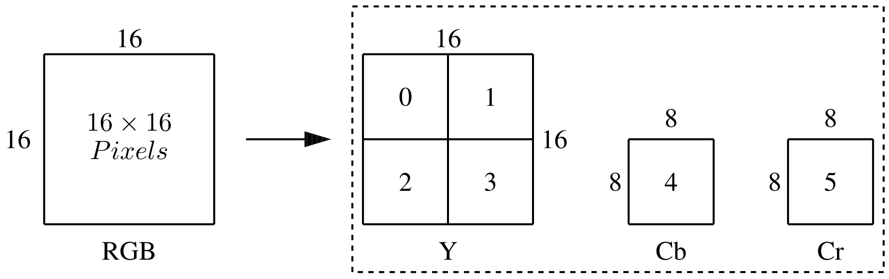


Figure 5-1. Macroblock of 4:2:0

5.2 Implementing RGB and 4:2:0 YCbCr Transformation for Video Frames

When implementing the conversion of RGB to YCbCr, we process the data in units of macroblocks; a macroblock has four 8×8 Y sample blocks and one 8×8 sample block for each of Cb and Cr samples. We also assume that a frame has an integral number of macroblocks. In the **4:2:0** YCbCr format, for each RGB pixel, we make a conversion for Y but we only make a conversion for Cb and Cr for every four RGB pixels (see Figure 3-8). Each group is formed by grouping 4 neighbouring pixels. For simplicity, when calculating the Cb, and Cr components, we simply use the upper left pixel of the four and ignore the other three. (Alternatively, one can take the average value of the four RGB pixel values when calculating the Cb and Cr values.) For convenience of programming, we define three **structs**: struct **RGB** holds the RGB values of a pixel, struct **YCbCr** holds the Ycbcr values of a pixel, and **YCbCr_MACRO** holds the YCbCr sample values of a macroblock of ($16 \times 16\ pixels$) of Figure 5-1. In C++, a **structs** is a **class** with public data members only. We put all these definitions in the header file "common.h", which is listed in Figure 5-2.

```
//common.h
//defines an RGB pixel
class RGB {
public:
  unsigned char R;             //0 - 255
  unsigned char G;             //0 - 255
  unsigned char B;             //0 - 255
};

class YCbCr {
public:
  unsigned char Y;             //0 - 255
  unsigned char Cb;            //0 - 255
  unsigned char Cr;            //0 - 255
};

//4:2:0 YCbCr Macroblock
class YCbCr_MACRO {
public:
  unsigned char Y[256];        //16x16 ( four 8x8 samples )
  unsigned char Cb[64];        //8x8
  unsigned char Cr[64];        //8x8
};

class RGBImage {
public:
  short width;                 //image width
  short height;                //image height
  unsigned char *ibuf;         //pointer to buffer
                               //  holding image data
};
```

Figure 5-2 Public classes (structs) for Processing Macro Blocks

We have learned in Chapter 3 how to convert an RGB pixel to YCbCr values using integer arithmetic. We define a function named **rgb2ycbcr** (RGB &a, YCbCr &b) to convert an RGB pixel a to a YCbCr pixel b and a function named **rgb2y**(RGB &a, unsigned char &y) to convert an RGB pixel a to a Y component y. Now, we need a function to convert an entire RGB macroblock to a

4:2:0 YCbCr macroblock, which consists of four 8×8 Y sample blocks, one 8×8 Cb sample block and one 8×8 Cr sample block. The following function **macroblock2ycbcr()**, listed in Figure 5-3 does the job; the input of it can be considered as a 16×16 two dimensional array, *macro16x16*[][] holding 16×16 RGB pixels; the output is a pointer to a **YCbCr_MACRO** struct defined in Figure 5-2, which holds the converted Y, Cb, Cr sample values with array Y[] holding the four Y sample blocks, and arrays Cb[] and Cr[] holding the sample blocks Cb and Cr respectively.

```
/*
  Convert an RGB macro block ( 16x16 ) to 4:2:0 YCbCr sample blocks
  (six 8x8 blocks).
*/
void macroblock2ycbcr( RGB *macro16x16,  YCbCr_MACRO *ycbcr_macro )
{
  int i, j, k, r;
  YCbCr ycb;

  r = k = 0;
  for ( i = 0; i < 16; ++i ) {
    for ( j = 0; j < 16; ++j ) {
      if ( !( i & 1 ) && !( j & 1 ) ) {  //1 Cb,Cr for 4 pixels
        rgb2ycbcr(macro16x16[r], ycb);//convert to Y,Cb,Cr values
        ycbcr_macro->Y[r] = ycb.Y;
        ycbcr_macro->Cb[k] = ycb.Cb;
        ycbcr_macro->Cr[k] = ycb.Cr;
        k++;
      } else {                    //only need Y for other 3 pixels
        rgb2y ( macro16x16[r], ycbcr_macro->Y[r] );
      }
      r++;                        //convert every pixel for Y
    }
  }
}
```

Figure 5-3 Function for converting an RGB macroblock to 4:2:0 YCbCr Sample Blocks

In Figure 5-3, the statement "*if (!(i & 1) && !(j & 1))* { " is true only when both *i* and *j* are even. This implies that it selects one pixel for a group of four neighbouring pixels as shown in Figure 3-8, and makes a conversion to Y, Cb, Cr; it makes a conversion of only the Y component for the other 3 pixels as we have considered the 4:2:0 YCbCr format. For example, the statement is true when

$(i, j) = (0, 0), (0, 2), ..., (2, 0), (2, 2), ..., (14, 14).$

We can similarly define a function, **ycbcr2macroblock()** to convert a YCbCr macroblock to an RGB macroblock. The following program, **rgb_ycc.cpp** of Listing 5-1 contains all the functions we need to convert video frames from RGB to YCbCr and back. It can be compiled to object code **rgb_ycc.o** by the command "g++ -c rgb_ycc.cpp".

Program Listing 5-1 (rgb_ycc.cpp) RGB-YCbCr Conversions

```
----------------------------------------------------------------------
/*
  Convert from RGB to YCbCr using  ITU-R recommendation BT.601.
     Y = 0.299R + 0.587G + 0.114B
    Cb = 0.564(B - Y ) + 0.5
    Cr = 0.713(R - Y ) + 0.5

  Integer arithmetic is used to speed up calculations.
  Note:
       2^16 = 65536
       kr = 0.299 = 19595 / 2^16
       kg = 0.587 = 38470 / 2^16
       Kb = 0.114 =  7471 / 2^16
       0.5 = 128 / 255
       0.564 = 36962 / 2^16
       .......
  Input: an RGB pixel
  Output: a YCbCr "pixel".
*/

void rgb2ycbcr( RGB &rgb, YCbCr &ycc )
{
  //coefs summed to 65536(1<<16), so Y is always within [0, 255]

  ycc.Y = (unsigned char)((19595 * rgb.R + 38470 * rgb.G +
              7471 * rgb.B ) >> 16);
  ycc.Cb = (unsigned char)((36962*(rgb.B - ycc.Y ) >> 16) + 128);
  ycc.Cr = (unsigned char)((46727*(rgb.R - ycc.Y ) >> 16) + 128);
}

//just convert an RGB pixel to Y component
void rgb2y( RGB &rgb, unsigned char  &y )
{
  y = (short)((19595*rgb.R + 38470*rgb.G + 7471*rgb.B ) >> 16);
}

//limit value to lie within [0,255]
void chop ( int & r, int &g, int &b )
{
    if ( r < 0 ) r = 0;
    else if ( r > 255 ) r = 255;
    if ( g < 0 ) g = 0;
    else if ( g > 255 ) g = 255;
    if ( b < 0 ) b = 0;
    else if ( b > 255 ) b = 255;
}

/*
  Convert from YCbCr to RGB domain. Using ITU-R standard:
     R = Y + 1.402Cr - 0.701
     G = Y - 0.714Cr - 0.344Cb + 0.529
     B = Y + 1.772Cb - 0.886
  Integer arithmetic is used to speed up calculations.
*/
void ycbcr2rgb( YCbCr &ycc, RGB &rgb )
{ int r, g, b;

  r = ( ycc.Y          + ( 91881 * ycc.Cr   >> 16 ) - 179 );
  g = ( ycc.Y  - (( 22544 * ycc.Cb + 46793 * ycc.Cr ) >> 16) + 135);
  b =  ( ycc.Y  + (116129 * ycc.Cb   >> 16 ) - 226 );
  chop ( r, g, b );           //enforce values to lie within [0,255]
  rgb.R = ( unsigned char ) r;
  rgb.G = ( unsigned char ) g;
  rgb.B = ( unsigned char ) b;
}

/*
  Convert an RGB macro block ( 16x16 ) to 4:2:0 YCbCr sample blocks
  ( six 8x8 blocks ).
*/
void macroblock2ycbcr ( RGB *macro16x16,  YCbCr_MACRO *ycbcr_macro )
{
  int i, j, k, r;
  YCbCr ycb;

  r = k = 0;
  for ( i = 0; i < 16; ++i ) {
    for ( j = 0; j < 16; ++j ) {
      if (!( i & 1 ) && !( j & 1 )){//one Cb, Cr for every 4 pixels
        rgb2ycbcr(macro16x16[r], ycb);//convert to Y, Cb, Cr values
        ycbcr_macro->Y[r] = ycb.Y;
        ycbcr_macro->Cb[k] = ycb.Cb;
        ycbcr_macro->Cr[k] = ycb.Cr;
        k++;
      } else {       //only need the Y component for other 3 pixels
        rgb2y ( macro16x16[r], ycbcr_macro->Y[r] );
      }
      r++;           //convert every pixel for Y
    }
  }
}

/*
  Convert the six 8x8 YCbCr sample blocks to RGB macroblock(16x16).
*/
void ycbcr2macroblock( YCbCr_MACRO *ycbcr_macro, RGB *macro16x16 )
{
  int i, j, k, r;
  short y;
  YCbCr ycb;
  r = k = 0;
  for ( i = 0; i < 16; ++i ) {
    for ( j = 0; j < 16; ++j ) {
      //one Cb, Cr has been saved for every 4 pixels
      if ( !( i & 1 ) && !( j & 1 ) ) {
        ycb.Y = ycbcr_macro->Y[r];
        ycb.Cb = ycbcr_macro->Cb[k];
        ycb.Cr = ycbcr_macro->Cr[k];
        ycbcr2rgb ( ycb, macro16x16[r]);
        k++;
      } else {
        ycb.Y = ycbcr_macro->Y[r];
        ycbcr2rgb( ycb, macro16x16[r] );
      }
      r++;
    }
  }
}
--------------------------------------------------------------------------
```

After we have implemented the functions to convert an RGB macroblock to a YCbCr macroblock and vice versa, we can utilize these functions to convert an image frame from RGB to YCbCr and save the data. As 4:2:0 format is used, the saved YCbCr data are only half as much as the original RGB data. In our discussions, the four Y sample blocks of a YCbCr macroblock is stored in the linear array Y[256] (Figure 5-2). In some cases, it is more convenient to separate the 256 Y samples into four sample blocks, each with size 64 (8×8). The relation between the indexes of the linear array and the four sample blocks, labeled 0, 1, 2, and 3 is shown in Figure 5-4 and the code that divides the 256 samples into four arrays is shown in the figure below.

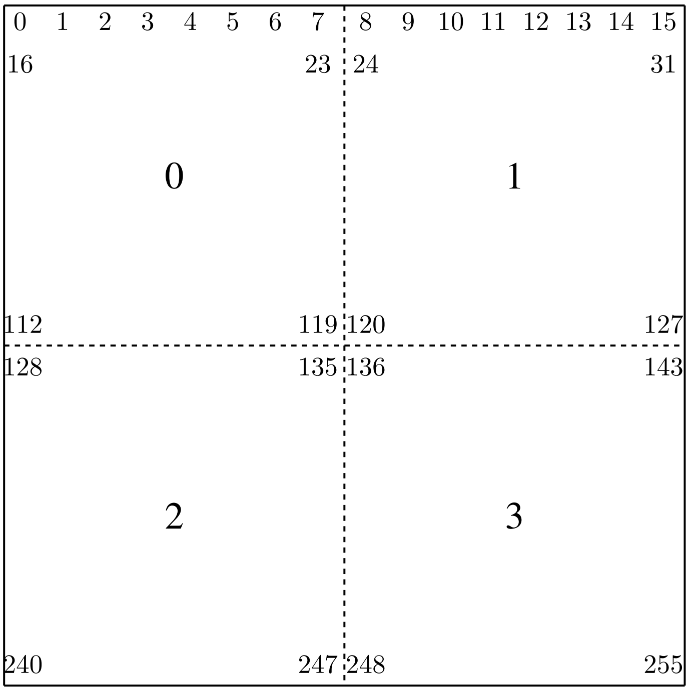


Figure 5-4. Y Sample Block Indexes

The following code shows how to separate a Y block of Figure 5-4 to four sublocks.

```
YCbCr_MACRO ycbcr_macro;
....
unsigned char Y4[4][64];
int start, k, r;

//start always points the beginning of a block
for(int b=0; b < 4; b++){
  if ( b < 2 )
    start = 8 * b;
  else
    start = 128 + 8 * ( b - 2 );
  k = start;
  r = 0
  for (int i = 0; i < 8; i++){ //one sample block
    for(int j=0; j < 8; j++)
      Y4[b][r++] = ycbcr_macro.Y[k+j];
    k += 16;                    //next row
  }
```

The program **encode.cpp**, shown in Listing 5-2 uses a similar code to save Y samples as four 8×8 blocks.

Program Listing 5-2

```
------------------------------------------------------------------------
/*
  encode.cpp
  Contains functions to convert an RGB frame to YCbCr and to save
  the converted data.
  Compile: g++ -c encode.cpp
*/
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "rgb_ycc.h"

//save one YCbCr macroblock.
void save_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpo )
{
  short block, i, j, k;
  unsigned char *py;

  //save four 8x8 Y sample blocks
  for ( block = 0; block < 4; block++ ) {
    if ( block < 2 )
      //points to beginning of block
      py = ( unsigned char * ) &ycbcr_macro->Y + 8*block;
    else
      py =(unsigned char *)&ycbcr_macro->Y+128+8*(block-2);
    for ( i = 0; i < 8; i++ ) { //one sample-block
      if ( i > 0 ) py += 16;    //advance py by 16( one row )
      for ( j = 0; j < 8; j++ ) {
        putc ( ( int )  *(py+j),fpo);   //save one byte of data
      }
    }
  }

  //save one 8x8 Cb block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      putc( ( int ) ycbcr_macro->Cb[k++], fpo );
    }
  }

  //save one 8x8 Cr block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      putc( ( int ) ycbcr_macro->Cr[k++], fpo );
    }
  }
}

/*
  Convert RGB to YCbCr and save the converted data.
```

```
*/
void encode ( RGBImage *image, FILE *fpo )
{
  short row, col, i, j, r;
  RGB macro16x16[256];    //16x16 pixel macroblock;24-bit RGB pixel
  YCbCr_MACRO ycbcr_macro;//macroblock for YCbCr samples
  RGB *p;                 //pointer to an RGB pixel
  static int nframe = 0;

  for ( row = 0; row < image->height; row += 16 ) {
    for ( col = 0; col < image->width; col += 16 ) {
      //points to beginning of macroblock
      p=(RGB *)image->ibuf+(row*image->width + col);
      r = 0;                    //note pointer arithmetic
      for ( i = 0; i < 16; ++i ) {
        for ( j = 0; j < 16; ++j ) {
          macro16x16[r++] = (RGB) *p++;
        }
        p += (image->width-16); //points to next row within macroblock
      }
      macroblock2ycbcr ( macro16x16,  &ycbcr_macro );//RGB to YCbCr
      save_yccblocks( &ycbcr_macro, fpo );  //save one YCbCr macroblock
    } //for col
  } //for row
}
-------------------------------------------------------------------------
```

In **encode.cpp** of Listing 5-2, the function **save_yccblocks()** saves one YCbCr macroblock in a file pointed by the the file pointer *fpo*; the function **encode()** makes use of **macroblock2ycbcr()** and **save_yccblocks()** to convert an RGB image or frame to YCbCr and saves the converted data in a file. Note that the standard C function putc() saves only the lower 8-bit of an integer and this is what we want; we can use getc() to read the byte back.

The corresponding code that converts a file of YCbCr data to RGB data is shown in **decode.cpp** of Listing 5-3.

Program Listing 5-3

```
-------------------------------------------------------------------------
/*
  decode.cpp
  Contains functions to read YCbCr data from a file and convert
  from YCbCr to RGB.
  Compile: g++ -c decode.cpp
*/
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "rgb_ycc.h"

/*
  Get YCbCr data from file pointed by fpi. Put the four 8x8 Y sample
  blocks, one 8x8 Cb sample block and one 8x8 Cr sample block into a
  struct ( class object) of YCbCr_MACRO.
  Return: number of bytes read from file and put in
          YCbCr_MACRO struct.
```

```
*/
int get_yccblocks( YCbCr_MACRO *ycbcr_macro, FILE *fpi )
{
  short r, row, col, i, j, k, n, block;
  short c;
  unsigned char *py;

  n = 0;
  //read data from file and put them in four 8x8 Y sample blocks
  for ( block = 0; block < 4; block++ ) {
    if ( block < 2 )
      //points to beginning of block
      py = ( unsigned char * ) &ycbcr_macro->Y + 8*block;
    else
      py = (unsigned char *)&ycbcr_macro->Y+128+8*(block-2);
    for ( i = 0; i < 8; i++ ) {       //one sample-block
      if ( i > 0 ) py += 16;          //advance py by 16 (one row)
      for ( j = 0; j < 8; j++ ) {
        if ( ( c = getc ( fpi ))  == EOF) //read one byte
              break;
        *(py + j) = (unsigned char) c; //save in YCbCr_MACRO struct
         n++;
      } //for j
    } //for i
  } //for block
  //now do that for 8x8 Cb block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      if ( ( c = getc ( fpi )) == EOF )
        break;
        ycbcr_macro->Cb[k++] = (unsigned char )c;
        n++;
    }
  }

  //now do that for 8x8 Cr block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      if ( ( c = getc ( fpi )) == EOF )
        break;
      ycbcr_macro->Cr[k++] = (unsigned char) c;
      n++;
    }
  }
  return n;                                //number of bytes read
}

/*
 *   Convert a YCbCr frame to an RGB frame.
 */
int decode_yccFrame ( RGBImage &image, FILE *fpi )
{
  short r, row, col, i, j, k, block;
  int n = 0;
  RGB macro16x16[256];      //16x16 pixel macroblock; 24-bit RGB pixel
  YCbCr_MACRO ycbcr_macro;//macroblock for YCbCr samples
  RGB *rgbp;                //pointer to an RGB pixel
  for ( row = 0; row < image.height; row += 16 ) {
    for ( col = 0; col < image.width; col += 16 ) {
      int m = get_yccblocks( &ycbcr_macro, fpi );
      if ( m <= 0 ) { printf("\nout of data\n"); return m;}
      n += m;
      ycbcr2macroblock( &ycbcr_macro, macro16x16 );
      //points to beginning of macroblock
      rgbp = ( RGB *)(image.ibuf)+(row*image.width+col);
      r = 0;
      for ( i = 0; i < 16; ++i ) {
        for ( j = 0; j < 16; ++j ) {
          *rgbp++ =  macro16x16[r++];
        }
        rgbp += (image.width - 16);//next row within macroblock
      }
    } //for col
  }  //for row
  return n;
}
----------------------------------------------------------------------
```

In **decode.cpp** of Listing 5-3, the function **get_yccblocks()** gets YCbCr data from a file pointed by file pointer *fpi*; the data are organized in macroblocks, consisting of four 8x8 Y sample blocks, one 8x8 Cb sample block and one 8x8 Cr sample block. The function saves the data into a struct (class object) of YCbCr_MACRO. The function **decode_yccFrame()** uses **get_yccblocks()** to convert the YCbCr data of an image or frame saved in a file to RGB and stores the RGB data in the buffer of a class object of RGBImage.

5.3 Testing Implementation Using PPM Image

We can test the implementation presented in section 5.2 using a PPM image, the format of which has been discussed in Chapter 4. It is the simplest portable format that one can have and does not have any compression. As pointed out before, the implementation of section 5.2 only works for images with height and width divisible by 16. If you obtain a PPM image with dimensions non-divisible by 16, you need to use the "convert" utility with the "-resize" option to change its dimensions before doing the test.

Again, we simplify our code by hard-coding the file names used for testing. We put the testing files in the directory "../data/", a child directory of the parent of the directory the testing programs reside. Suppose the testing file is called "beach.ppm". All we want to do is to read the RGB data of "beach.ppm", convert them to YCbCr and save the YCbCr macroblocks in the file "beach.ycc". In saving the YCbCr macroblocks, we also need to save the image dimensions for decoding. We employ a very simple format for our ".ycc" file; the first 8 bytes contain the header text, "YCbCr420"; the next two bytes contain the image width followed by another two bytes of image height; data start from the thirteenth byte. We then read the YCbCr macroblocks back from "beach.ycc" into a buffer and convert the YCbCr data to RGB. We save the recovered RGB data in the file "beach1.ppm". The testing program **test_encode_ppm.cpp** that performs these tasks is listed in Listing 5-4.

Program Listing 5-4

```
------------------------------------------------------------------------
/*
 * test_encode_ppm.cpp
 * Program to test integer implementations of RGB-YCbCr conversions
 * and the use of macroblocks to compress data. PPM files are used
 * for testing. It reads "../data/beach.ppm" RGB data, converts
 * them to 4:2:0 YCbCr macroblocks which will be saved in file
 * "../data/beach.ycc"; it then reads back the YCbCr macroblocks
 * from "../data/beach.ycc", converts them to RGB data and save
 * the data in "../data/beach1.ppm".  PPM files can be viewed using
 * "xview".
 * Compile: g++ -o test_encode_ppm test_encode_ppm.cpp rgb_ycc.o \
 *                   encode.o decode.o
 * Execute: ./test_encode_ppm
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "common.h"

//A public class is the same as a 'struct'
class CImage {
public:
  unsigned char red;
  unsigned char green;
  unsigned char blue;
};

void encode ( RGBImage *image, FILE *fpo );
int decode_yccFrame ( RGBImage &image, FILE *fpi );

/*
 * Create PPM header from image width and height. "P6" format used.
 * PPM header returned in integer array ppmh[].
 */
void make_ppm_header ( int ppmh[], int width, int height )
{
  //standard header data, 'P' = 0x50, '6' = 0x36, '\n' = 0x0A
  int ca[] = {0x50, 0x36, 0x0A,                          //"P6"
      //image width=260,height=288
      0x33, 0x36, 0x30, 0x20, 0x32, 0x38, 0x38, 0x0A,
      0x32, 0x35, 0x35, 0x0A }; //color levels/pixel=256
  //only have to change width and height
  char temp[10], k;
  sprintf(temp, "%3d", width );         //width in ascii code
  k = 0;
  for ( int i = 3; i <= 5; ++i )        //replace width
    ca[i] = temp[k++];
  sprintf(temp, "%3d", height );        //height in ascii code
  k = 0;
  for ( int i = 7; i <=9; ++i )         //replace height
    ca[i] = temp[k++];
  for ( int i = 0; i < 15; ++i )        //form header
    ppmh[i] = ca[i];
}

void save_ppmdata ( RGBImage &image, FILE *fp )
{
  int size = 3 * image.width * image.height;

  for ( int i = 0; i < size; ++i )
    putc ( image.ibuf[i], fp ) ;
}

void ppm_read_comments ( FILE *fp )
{
  int c;
  while ( (c = getc ( fp ) )  == '#' ) {
    while ( getc( fp ) != '\n' )
;
  }
  ungetc ( c, fp );
}

class ppm_error
{
  public:
    ppm_error() {
      printf("\nIncorrect PPM format!\n");
      exit ( 1 );
    }
};

void write_ycc_header( short width, short height, FILE *fpo )
{
  char header[] = { 'Y', 'C', 'b', 'C', 'r', '4', '2', '0' };
  for ( int i = 0; i < 8; ++i )
    putc ( header[i], fpo );
  char *p;
  p = ( char *) &width;      //points to lower byte of width
  putc ( *p++,  fpo );       //save lower byte of width
  putc ( *p,  fpo );         //save upper byte of width
  p = ( char *) &height;     //points to lower byte of height
  putc ( *p++,  fpo );       //save lower byte of height
  putc ( *p,  fpo );         //save upper byte of height
}

int read_ycc_header( short &width, short &height, FILE *fpi )
{
  char header[9];
  for ( int i = 0; i < 8; ++i )
    header[i] = (char) getc ( fpi );
  if ( strncmp ( header, "YCbCr420", 8 ) )
    return -1;
  char *p;
  p = ( char *) &width;       //read the width
  *p++ = getc ( fpi );
  *p = getc ( fpi );
  p = ( char *) &height;     //read the height
  *p++ = getc ( fpi );
  *p = getc ( fpi );

  return 1;
}

int main()
{
  FILE *fp;
  int c;
  fp = fopen ("../data/beach.ppm", "rb");//PPM file for testing
  RGBImage image;

  ppm_read_comments ( fp );              //read comments
  char temp[100];
  fscanf ( fp, "%2s", temp );
  temp[3] = 0;
  if ( strncmp ( temp, "P6", 2 ) )
    throw ppm_error();
  ppm_read_comments ( fp );
  fscanf ( fp, "%d", &image.width );
  ppm_read_comments ( fp );
  fscanf ( fp, "%d", &image.height );
  ppm_read_comments ( fp );
  int colorlevels;
  fscanf ( fp, "%d", &colorlevels );
  ppm_read_comments ( fp );
  while ((c = getc ( fp )) == '\n');//get rid of extra line returns
  ungetc ( c ,fp );

  if ( image.width % 16 != 0 || image.height % 16 != 0 ) {
    printf("\nProgram only works for image dimensions divisible \
           by 16.\n \
           Use 'convert' utility to change image dimension!\n");
    return 1;
  }

  int isize;
  isize = image.width * image.height;
  //allocate memory to hold RGB data
  image.ibuf = (unsigned char *)malloc (3*isize);
  fread ( image.ibuf,  3, isize, fp );
  fclose ( fp );

  //encode RGB data in YCbCr 4:2:0 format and save in "beach.ycc"
  fp = fopen ( "../data/beach.ycc", "wb" );
  write_ycc_header ( image.width, image.height, fp );
  encode ( &image, fp );
  delete image.ibuf;                 //remove the image buffer
  fclose ( fp );

  //read the YCbCr data back from "beach.ycc" and convert to RGB
  fp = fopen ( "../data/beach.ycc", "rb" );
  if ( read_ycc_header ( image.width, image.height, fp ) == -1 ){
    printf("\nNot YCC File\n");
    return 1;
  }
  isize = image.width * image.height;
  printf("\nImage width:%d, height:%d\n",image.width,image.height);
  //allocate memory to hold image
  image.ibuf = ( unsigned char *) malloc ( 3 * isize );
  decode_yccFrame ( image, fp );
  fclose ( fp );

  //now save the decoded data in ppm again
  fp=fopen("../data/beach1.ppm","wb");//output PPM file for testing
  int ppmh[20];                        //PPM header*
  make_ppm_header ( ppmh, image.width, image.height );
  for ( int i = 0; i < 15; ++i )       //save PPM header
    putc ( ppmh[i], fp );
  save_ppmdata ( image, fp );          //save RGB data
  fclose ( fp );
  delete image.ibuf;                   //deallocate memory

  return 0;
}
------------------------------------------------------------------------
```

To generate an executable, we have to link this file to the object files discussed in section 5.2 using a command similar to the following:

```
$g++ -o test_encode_ppm test_encode_ppm.cpp rgb_ycc.o encode.o decode.o
```

The program "test_encode_ppm" uses some of the functions discussed in Chapter 4 to read and write PPM files. When it is executed, it does the following:

1. reads RGB data from "../data/beach.ppm",
2. saves YCbCr data in "../data/beach.ycc", and
3. saves reconstructed RGB data in "../data/beach1.ppm".

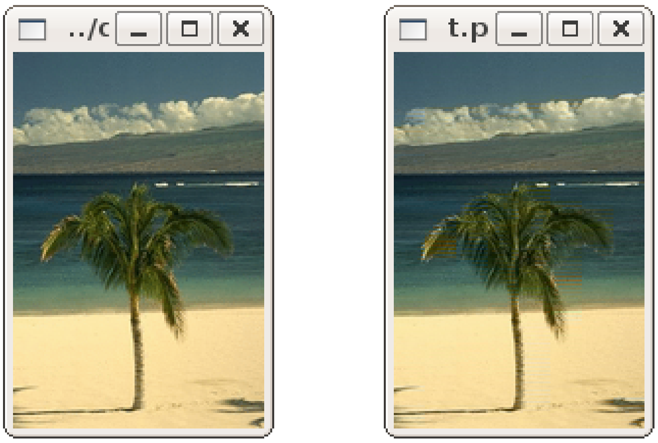
You can view the PPM files using "xview"; the command "xview ../data/beach.ppm" displays the original RGB image and the command "xview ../data/beach1.ppm" displays the recovered RGB image.

You should find that the two images almost look identical to each other even though we have compressed "beach.ppm" to "beach.ycc" by a factor of two. If you want to find out the file sizes, you can issue the command "ls -l ../data/beach*". Upon executing this command, you should see a display similar to the following:

```
73743  ../data/beach1.ppm
73743  ../data/beach.ppm
36876  ../data/beach.ycc
```

The first column indicates the file sizes in bytes. As you can see, files "beach.ppm" and "beach1.ppm" have identical size but the file "beach.ycc" that contains YCbCr data is only half the size of the file "beach.ppm" that contains RGB data. You can examine the image using the command like **display ../data/beach1.ppm**.

Figure 5 presents the images we have used and generated in our experiment; the original RGB image (beach.ppm) is shown in Figure 5-5a and the restored RGB image (beach1.ppm) is shown in Figure 5-5b.



a) Original b) Recovered

Figure 5-5 Original and Recovered Images in 4:2:0 Transformations