An Introduction to Digital Video Data Compression in Java

Fore June

Chapter 5 Macroblocks

5.1 Introduction

In general, a PC image or a frame with moderate size 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:

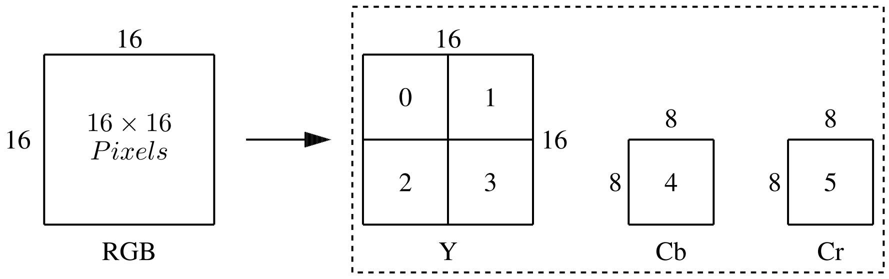

Figure 5-1. Macroblock of 4:2:0

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, one 8×8 Cb sample block and one 8×8 Cr sample block. We 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 3-5). Each group of four pixels 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.)

Before discussing the implementation of RGB-YCbCr transformations, we would like to highlight some programming features of java that some readers may overlook or misundestand.

Some Notes on Java Programming

Firstly, java always passes parameters by values; it does not support pass-by-reference. This means that we cannot change the value of a variable by passing it as an argument to a function.

Secondly, a java object variable is analogous to an object pointer in C++. For example, suppose we have defined a **Pixel** class:

```
class Pixel {
  int red;
  int green;
  int blue;
};
```

The java code,

```
Pixel aPixel = new Pixel();       //java
```

is the same as the C++ code,

```
Pixel *aPixel = new Pixel();       //C++
```

Basically, we can always regard a java object variable as a pointer (address) pointing to the object. Though we cannot change the value of a function parameter, we can change the state of the object that a function parameter points at. For example, the following function changes the value of *red*:

```
void setPixel ( Pixel p ) {
  p.red = 100;
}
```

The function sets the value of *red* of the **Pixel** object pointed by *p* to 100. On the other hand, the following piece of code does not make any change to the object.

```
void setPixel ( Pixel p ) {
  p = new Pixel();
  p.red = 100;
}
```

This is because in the function, *p* is assigned a new value, pointing to a local object. This new local pointer value will not be passed back to the calling function.

In general, most parts of a java program written for manipulating objects involve assigning pointer values rather than object values. For example, consider a statement such as the following:

```
x = objectVariable;
```

In C++, this means copying the object denoted by *objectVariable* to variable *x*. They then represent two different objects. Changing x will not affect *objectVariable*. However, in java, **no object-copying** occurs; the statement simply means assigning the pointer *objectVariable* to *x*. When you change the object referenced by *x*, you also change the object referenced by *objectVariable* as both of them point to the same object.

Thirdly, unlike C++, when we declare an array of objects, java does not create the objects automatically. Rather, it defines an array of pointers initialized to NULL. For example, if we want to create an array of Pixel objects, we must use the **new** operator to do the instantiation like the following:

```
class PixelArray {
  Pixel [] pa = new Pixel[16];//create an array of NULL pointers
  PixelArray(){               //constructor
    for ( int i = 0; i < 16; ++i ) //instantiation
      pa[i] = new Pixel();
  }
}
```

Fourthly, the **round**() function of the java Math library does not work properly like that of C++. Java rounds 0.5 to 1 but it rounds -0.5 to 0 rather than -1. That is, Math.round (0.5) yields 1 but Math.round (-0.5) yields 0. This asymmetry in handling positive and negative numbers may cause problems in programming in the future. Readers may need to add some extra coding to correct this defect in their programs.

Finally, we would like to explain briefly the difference between a class and an object. We believe most readers understand the difference but there may be a few who come from a different programming background and may not be familiar with terminologies of object-oriented programming (OOP). In short, a class refers to the code of a structure that performs certain tasks. A class contains both data (referred to as attributes), and functions (referred to as methods). An object is an instantiation of a class and is instantiated by the **new** keyword. This is in analogy of building a house. The blueprint that specifies the details of building a house corresponds to a class. A house built based on the blueprint corresponds to an object. Building a house consumes resources such as glass, wood and stone. In the same way, creating an object requires resources such as memory and files. We can build more than one house using the same blueprint. In a similar sense, we can create more than one object based on the same class. When we use a class name in a description, we may refer to either the class or an object depending on the context of description. Also, java functions and methods mean the same thing in this book.

Implementation

For convenience of programming, we define four classes: the object of class **RGB** holds the RGB values of a pixel, **YCbCr** holds the YCbCr values of a pixel, and **RGB_MACRO** and **YCbCr_MACRO** hold the RGB and YCbCr sample values of a macroblock ($16 \times 16\ pixels$) of Figure 5-1 respectively. We put all these definitions in the file "common.java", which is listed in Figure 5-2 below.

```
//common.java
class RGB {   //defines an RGB pixel
  int R;      //0 - 255
  int G;      //0 - 255
  int B;      //0 - 255
}
class YCbCr {
  int Y;      //0 - 255
  int Cb;     //0 - 255
  int Cr;     //0 - 255
}
class RGB_MACRO {  //16x16 RGB block
  RGB [] rgb;
  RGB_MACRO(){     //constructor
    rgb = new RGB[256];
    //allocate memory
    for ( int i = 0; i < 256; ++i )
      rgb[i] = new RGB();
  }
}
class YCbCr_MACRO {         //4:2:0 YCbCr Macroblock
  int [] Y = new int[256];  //16x16 (four 8x8 samples)
  int [] Cb  = new int[64]; //8x8
  int [] Cr  = new int[64]; //8x8
}
class RGBImage {
  int width;                //image width
  int height;               //image height
  RGB  [] ibuf;             //image data buffer
  //constructor
  RGBImage ( int w, int h ) {
    width = w;
    height = h;
    ibuf = new RGB[width*height];//image data buffer
    for ( int i = 0; i < width * height; ++i )
      ibuf[i] = new RGB();
  }
}
```

Figure 5-2 Public classes 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) to convert an RGB pixel a to a Y component and returns its value as an integer. 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 is a java object variable (pointer) pointing to an **RGB_MACRO** object containing the data of a 16×16 RGB macroblock; the function converts the RGB values to YCbCr values and put the data in a **YCbCr_MACRO** object pointed by the object variable of the second input parameter of the function and thus the converted values will be sent back to the calling function via this parameter.

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

    r = k = 0;
    for ( i = 0; i < 16; ++i ) {
      for ( j = 0; j < 16; ++j ) {
        //need one Cb, Cr for every 4 pixels
        if ( ( i & 1 ) == 0 && ( j & 1 ) == 0 ) {
          //convert to Y and Cb, Cr values
          rgb2ycbcr ( rgb_macro.rgb[r], ycc );
          ycbcr_macro.Y[r] = ycc.Y;
          ycbcr_macro.Cb[k] = ycc.Cb;
          ycbcr_macro.Cr[k] = ycc.Cr;
          k++;
        } else { //only need Y component for other 3 pixels
          ycbcr_macro.Y[r] = rgb2y ( rgb_macro.rgb[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) == 0 && (j & 1) == 0)** { " is true only when both *i* and *j* are even. This implies that it selects one pixel from a group of four neighbouring pixels as shown in Figure 3-5, 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, **RgbYcc.java** of Listing 5-1 contains all the functions we need to convert video frames from RGB to YCbCr and back. It can be compiled with the command "javac RgbYcc.java".

Program Listing 5-1 Conversions between RGB and YCbCr

```
/*
  RgbYcc.java
  Class for converting RGB to YCbCr and vice versa.
*/

/*
  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
       0.713 = 46727 / 2^16
       1.402 = 91881 / 2^16
       0.701 = 135 / 255
       0.714 = 46793 / 2^16
       0.344 = 22544 / 2^16
       0.529 = 34668 / 2^16
       1.772 = 116129 / 2^16
       0.886 = 226 / 255
*/

import java.io.*;

class RgbYcc {
  public void rgb2ycbcr( RGB rgb, YCbCr ycc )
  {
    //coefs summed to 65536 (1 << 16), so Y is always within [0, 255]
    ycc.Y = (19595 * rgb.R + 38470 * rgb.G + 7471 * rgb.B ) >> 16;
    ycc.Cb = ( 36962 * ( rgb.B - ycc.Y ) >> 16 ) + 128 ;
    ycc.Cr = ( 46727 * ( rgb.R - ycc.Y )  >> 16 ) + 128 ;
  }

  //just convert an RGB pixel to Y component
  public int rgb2y( RGB rgb )
  {
    int y;

    y = ( 19595 * rgb.R + 38470 * rgb.G + 7471 * rgb.B ) >> 16;
    return y;
  }

  //limit value to lie within [0,255]
  public RGB chop ( RGB rgb )
  {
    if (rgb.R < 0 || rgb.B < 0 || rgb.B < 0 ||
          rgb.R > 255 || rgb.B > 255 || rgb.G > 255 )
      if ( rgb.R < 0 ) rgb.R = 0;
      else if ( rgb.R > 255 ) rgb.R = 255;
      if ( rgb.G < 0 ) rgb.G = 0;
      else if ( rgb.G > 255 ) rgb.G = 255;
      if ( rgb.B < 0 ) rgb.B = 0;
      else if ( rgb.B > 255 ) rgb.B = 255;

    return rgb;
  }

/*
  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.
*/

  public void ycbcr2rgb( YCbCr ycc, RGB rgb )
  {
```

```
  rgb.R = ycc.Y           + ( 91881 * ycc.Cr   >> 16 ) - 179;
  rgb.G = ycc.Y -(( 22544 * ycc.Cb + 46793 * ycc.Cr ) >> 16) + 135;
  rgb.B = ycc.Y  + (116129 * ycc.Cb   >> 16 ) - 226;

  rgb = chop ( rgb );     //enforce values to lie within [0,255]
}

/*
  Convert an RGB macro block ( 16x16 ) to
  4:2:0 YCbCr sample blocks ( six 8x8 blocks ).
*/

void macroblock2ycbcr ( RGB_MACRO rgb_macro, YCbCr_MACRO ycbcr_macro )
{
  int i, j, k, r;
  YCbCr ycc = new YCbCr();

  r = k = 0;
  for ( i = 0; i < 16; ++i ) {
    for ( j = 0; j < 16; ++j ) {
      //need one Cb, Cr for every 4 pixels
      if ( ( i & 1 ) == 0 && ( j & 1 ) == 0 ) {
        //convert to Y and Cb, Cr values
        rgb2ycbcr ( rgb_macro.rgb[r], ycc );
        ycbcr_macro.Y[r] = ycc.Y;
        ycbcr_macro.Cb[k] = ycc.Cb;
        ycbcr_macro.Cr[k] = ycc.Cr;
        k++;
      } else { //only need the Y component for other 3 pixels
        ycbcr_macro.Y[r] = rgb2y ( rgb_macro.rgb[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_MACRO rgb_macro )
{
  int i, j, k, r;
  YCbCr ycc = new YCbCr();

  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 ) == 0 && ( j & 1 ) == 0 ) {
        ycc.Y = ycbcr_macro.Y[r];
        ycc.Cb = ycbcr_macro.Cb[k];
        ycc.Cr = ycbcr_macro.Cr[k];
        k++;
      } else {
        ycc.Y = ycbcr_macro.Y[r];
        ycc.Cb = ycbcr_macro.Cb[k];
        ycc.Cr = ycbcr_macro.Cr[k];
      }
      ycbcr2rgb ( ycc, rgb_macro.rgb[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 are 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; the following piece of code shows how to separate the data into four 8×8 blocks:

```
int b, i, j, k, r;

//one macroblock has 256 Y samples
byte [] Yblock = new byte[256];

 r = 0;
//save four 8x8 Y sample blocks
for ( b = 0; b < 4; b++ ) {
  if ( b < 2 )
    k = 8 * b;                //points to beginning of block
  else
    k = 128 + 8 * ( b - 2 );  //points to beginning of block
  for ( i = 0; i < 8; i++ ) { //one sample-block
    if ( i > 0 ) k += 16;     //advance k by 16 (one row)
    for ( j = 0; j < 8; j++ ) {
      Yblock[r] = ( byte ) ycbcr_macro.Y[k+j];
      r++;
    }
  }
}
```

The array *Yblock* saves the Y sample blocks in the order of block 0, 1, 2 and 3 as shown in Figure 5-4. In the code, *Yblock* is a **byte** array, which prepares us to write the data to a file conveniently. On the other hand, *ycbcr_macro.Y* is a 32-bit integer array and we know that *Y* is always positive and lies in the range [0, 255]. However, in java, a **byte** is an 8-bit signed value. This means that any positive value larger than 127 and smaller than 256 will become a negative number when we convert it from an **int** to a **byte**. Will this cause error in the program? The answer is **no** because in a modern computer, any number is represented as a binary number. It is the way that we interpret a binary number that gives us a signed or an unsigned value. For example, an **int** of value 255 is represented by the 32-bit number 00000000000000000000000011111111. When we cast this to a **byte**, we take the lower eight bits, 11111111. If we interpret 11111111 as a signed-number, we obtain the value -1 but if we interpret it as an unsigned-number, we obtain the value 255. Regardless of our interpretation, we are processing the binary number 11111111. So if we save this binary number in a file and read it back, we always get back the same binary number. Of course, we need to be careful in our coding so that we will not change the 8-bit number 11111111 into the 32-bit binary number 11111111111111111111111111111111, which also represents the value of -1.

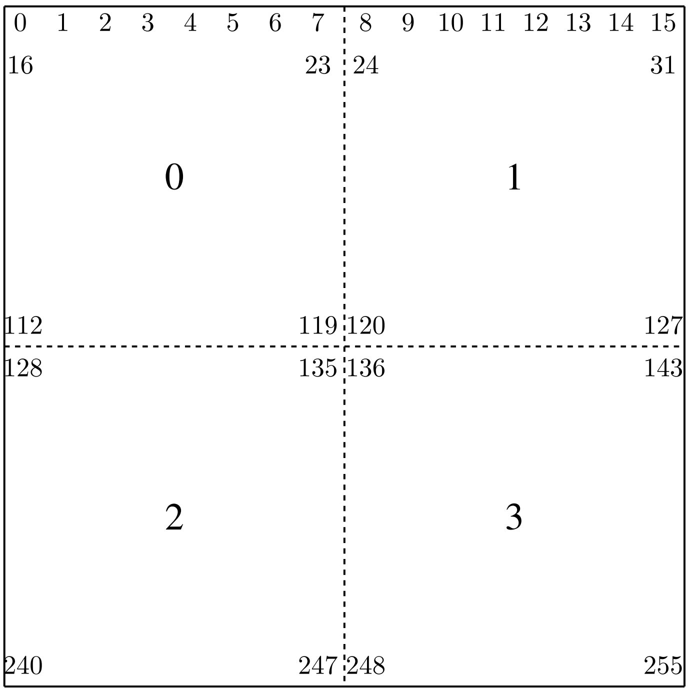

Figure 5-4 Y Sample Block Indexes

Program Listing 5-2 below shows the program **Encode.java** that contains the complete code of the class **Encode** that has functions to convert an RGB frame to YCbCr and to save the converted data in a file. The class only has two member functions, namely, **save_yccblocks()** and **encode()**. The function **save_yccblock()** saves one YCbCr block in the specified file; the function **encode()** makes use of **save_yccblocks()** and other functions of **RgbYcc** to convert a frame of RGB data to YCbCr and save the converted data in the file.

Program Listing 5-2 Encode Class

```
/*
  Encode.java
  Contains functions that convert an RGB frame to YCbCr and save the
  converted data.
*/

import java.io.*;

class Encode {
  //save one YCbCr macroblock.
  public void save_yccblocks(YCbCr_MACRO ycbcr_macro,DataOutputStream out)
  {
```

```
  int b, i, j, k, r;

  //one macroblock has 256 Y samples
  byte [] Yblock = new byte[256];

   r = 0;
  //save four 8x8 Y sample blocks
  for ( b = 0; b < 4; b++ ) {
    if ( b < 2 )
      k = 8 * b;                //points to beginning of block
    else
      k = 128 + 8 * ( b - 2 );  //points to beginning of block
    for ( i = 0; i < 8; i++ ) { //one sample-block
      if ( i > 0 ) k += 16;     //advance k by 16 (length of 1 row)
      for ( j = 0; j < 8; j++ ) {
        Yblock[r] = ( byte ) ycbcr_macro.Y[k+j];
        r++;
      }
    }
  }

  //save one 8x8 Cb block
  byte [] Cb_block = new byte[64];
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      Cb_block[k] = ( byte ) ycbcr_macro.Cb[k];
      k++;
    }
  }

  //save one 8x8 Cr block
  byte [] Cr_block = new byte[64];
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      Cr_block[k] = ( byte ) ycbcr_macro.Cr[k];
      k++;
    }
  }

  //save 4 Y sample blocks, one Cb block, one Cr block in file
  try {
    out.write ( Yblock );
    out.write ( Cb_block );
    out.write ( Cr_block );
  } catch (IOException e) {
    e.printStackTrace();
    System.exit(0);
  }
}

/*
  Convert 1 frome of RGB to YCbCr and save the converted data.
*/
public void encode ( RGBImage image, DataOutputStream out )
{
  int row, col, i, j, k, r;
  RGB_MACRO rgb_macro = new RGB_MACRO();
  //macroblock for YCbCr samples
```

```
    YCbCr_MACRO ycbcr_macro = new YCbCr_MACRO();
    RgbYcc rgbycc = new RgbYcc ();

    for ( row = 0; row < image.height; row += 16 ) {
      for ( col = 0; col < image.width; col += 16 ) {
        k = row * image.width + col;
        r = 0;
        for ( i = 0; i < 16; ++i ) {
  for ( j = 0; j < 16; ++j )
    rgb_macro.rgb[r++] = image.ibuf[k++];
  k += ( image.width - 16 );  //next row within macroblock
        }
        //convert from RGB to YCbCr
        rgbycc.macroblock2ycbcr( rgb_macro, ycbcr_macro );

        //save one YCbCr macroblock
        save_yccblocks( ycbcr_macro, out );
      } //for col
    } //for row
  }
}
```

Note that in **Encode.java** of Listing 5-2, the function **save_yccblocks()** uses **out.write()** to send 8-bit bytes to the file. In the decoding process, we will use something like **in.read()** to read one 8-bit byte back but the byte read is returned as a 32-bit integer and thus the value lies in the range [0, 255].

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

Program Listing 5-3 Decode Class

```
/*
  Decode.java
  Contains functions to read YCbCr data from a file and convert from YCbCr
  to RGB.
*/
import java.io.*;

class Decode {
 /*
  Get YCbCr data from file pointed by in. Put the four 8x8 Y sample blocks,
  one 8x8 Cb sample block and one 8x8 Cr sample block into a class object
  of YCbCr_MACRO.
  Return: number of bytes read from file.
 */
  public int get_yccblocks( YCbCr_MACRO ycbcr_macro, DataInputStream in )
  {
    int r, row, col, i, j, k, n, b, c;

    byte abyte;

    n = 0;
    //read data from file and put them in four 8x8 Y sample blocks
    for ( b = 0; b < 4; b++ ) {
      if ( b < 2 )
        k = 8 * b;                //points to beginning of block
      else
        k = 128 + 8 * ( b - 2 );  //points to beginning of block
      for ( i = 0; i < 8; i++ ) { //one sample-block
        if ( i > 0 ) k += 16;     //advance by 1 row of macroblock
        for ( j = 0; j < 8; j++ ){
try {
          if ( ( c = in.read() ) == -1 )   //read one byte
            break;
          ycbcr_macro.Y[k+j] = c;
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
        }
         n++;
      } //for j
    } //for i
  } //for b

  //now do that for 8x8 Cb block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      try {
        if ( ( c = in.read() ) == -1 )     //read one byte
          break;
        ycbcr_macro.Cb[k++] = c;
      } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
      }
      n++;
    }
  }

  //now do that for 8x8 Cr block
  k = 0;
  for ( i = 0; i < 8; ++i ) {
    for ( j = 0; j < 8; ++j ) {
      try {
        if ( ( c = in.read() ) == -1 )     //read one byte
          break;
        ycbcr_macro.Cr[k++] = c;
      } catch (IOException e) {
        e.printStackTrace();
        System.exit(0);
      }
      n++;
    }
  }
  return n;           //number of bytes read
}

/*   Read in YCbCr data from file.
 *   Convert a YCbCr frame to an RGB frame.
 *   Return RGB data via parameter image.
 */
public int decode_yccFrame ( RGBImage image, DataInputStream in )
{
  int r, row, col, i, j, k, block;
  int n = 0;
```

```
    //16x16 pixel macroblock; assume 24-bit for each RGB pixel
    RGB_MACRO rgb_macro = new RGB_MACRO();
    YCbCr_MACRO ycbcr_macro = new YCbCr_MACRO();
    RgbYcc rgbycc = new RgbYcc();
    for ( row = 0; row < image.height; row += 16 ) {
      for ( col = 0; col < image.width; col += 16 ) {
        int m = get_yccblocks( ycbcr_macro, in );
        if ( m <= 0 ) { System.out.printf("\nout of data\n"); return m;}
        n += m;
        rgbycc.ycbcr2macroblock( ycbcr_macro, rgb_macro );
        k = (row * image.width + col); //points to macroblock beginning
        r = 0;
        for ( i = 0; i < 16; ++i ) {
          for ( j = 0; j < 16; ++j ) {
            image.ibuf[k].R = rgb_macro.rgb[r].R;
            image.ibuf[k].G = rgb_macro.rgb[r].G;
            image.ibuf[k].B = rgb_macro.rgb[r].B;
            k++;  r++;
          }
          k += (image.width - 16);   //points to next row of macroblock
        }
      } //for col
    }  //for row
    return n;  //number of bytes read
  }
}
```

In **Decode.java** of Listing 5-3, the function **get_yccblocks()** gets YCbCr data from a file pointed by *in*; the data are organized in macroblocks, consisting of four 8×8 Y sample blocks, one 8×8 Cb sample block and one 8×8 Cr sample block. The function saves the data in a class object of YCbCr_MACRO and returns the data via the parameter *ycbcr_macro*. The function **decode_yccFrame()** uses **get_yccblocks()** to convert the YCbCr data of an image or frame to RGB, stores the RGB data in the buffer of a class object of RGBImage and returns the data via the first function parameter *image*.

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 5-3.

Program Listing 5-3

```
/*
 * Test_encode_ppm.java
 * Code testing the encoding and decoding of an RGB frame to and
 * from YCbCr format. The original RGB data are saved in a PPM
 * file.  The restored RGB data are saved in another PPM file.
*/
import java.io.*;
import java.awt.Frame;
import java.awt.image.*;
import javax.media.jai.JAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import javax.media.jai.widget.ScrollingImagePanel;
import com.sun.media.jai.codec.PNMEncodeParam;

public class Test_encode_ppm {
  private static byte [] header = {'Y','C','b','C','r','4','2','0'};

  public static void write_ycc_header(int width, int height,
                                              DataOutputStream out)
  {
    try {
       out.write ( header );
       out.writeInt ( width );
       out.writeInt ( height );
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }
  }

  public static int read_ycc_header( RGBImage rgbimage,  DataInputStream in)
  {
    byte [] bytes = new byte[header.length];
    try {
      in.read ( bytes );
      rgbimage.width  = in.readInt ();
      rgbimage.height = in.readInt ();
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }
    for ( int i = 0; i < header.length; ++i )
      if ( bytes[i] != header[i] )
        return -1;      //wrong header

    return 1;
  }

  public static void main(String[] args) throws InterruptedException {
    if (args.length < 3) {
      System.out.println("Usage: java " + "Test_encode_ppm" +
        " input_ppm_filename output_ycc_filename recoverd_ppm_filename\n" +
        "e.g. java Test_encode_ppm ../data/beach.ppm t.ycc t.ppm");
      System.exit(-1);
    }

    /*
     * Create an input stream from the specified file name
     * to be used with the file decoding operator.
     */
     FileSeekableStream stream = null;
     try {
       stream = new FileSeekableStream(args[0]);
     } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
     }

    /* Create an operator to decode the image file. */
    RenderedOp image = JAI.create("stream", stream);

    /* Get the width and height of image. */
    int width = image.getWidth();
    int height = image.getHeight();

    if ( width % 16 != 0 || height % 16 != 0 ) {
      System.out.println("Program only works for image dimensions divisible");
      System.out.println("by 16. Use 'convert' to change image dimension.");
      System.exit(1);
    }
    int [] samples = new int[3*width*height];

    Raster ras = image.getData();
    //save pixel RGB data in samples[]
    ras.getPixels( 0, 0, width, height, samples );
    RGBImage rgbimage = new RGBImage( width, height );
    //copy image data to RGBImage object buffer
    int isize = width * height;
    for ( int i = 0, k = 0; i < isize; ++i, k+=3 ) {
      rgbimage.ibuf[i].R =  samples[k];
      rgbimage.ibuf[i].G = samples[k+1];
      rgbimage.ibuf[i].B = samples[k+2];
    }

    try {
      File f = new File ( args[1] );
      OutputStream o = new FileOutputStream( f );
      DataOutputStream out = new DataOutputStream ( o );
      write_ycc_header ( width, height, out );
      Encode enc = new Encode ();

      //save encoded data ( YCbCr ) in file specified by args[1]
      enc.encode ( rgbimage, out );
      out.close();
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }

    System.out.printf("\nEncoding done, YCbCr data saved in %s\n",args[1]);

    //read the YCbCr data back from file args[1] and convert to RGB
```

```
    DataInputStream in;
    Decode ycc_decoder = new Decode ();
    try {
      File f = new File ( args[1] );
      InputStream ins = new FileInputStream( f );
      in = new DataInputStream ( ins );
      if ( read_ycc_header ( rgbimage,  in ) == -1 ){
        System.out.println("Not YCC File");
        return;
      }
      /*
        Decode data: convert YCbCr data from file args[1] to RGB.
        Restored RGB data returned via parameter rgbimage.
      */
      ycc_decoder.decode_yccFrame (rgbimage, in);
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }

    /*
      Now save RGB data in PPM format in the file specified by args[2].
    */
    isize = rgbimage.width * rgbimage.height;
    byte [] P6 = { 'P', '6', '\n' };
    byte [] colorLevels = { '2', '5', '5', '\n' };
    String sw = Integer.toString ( rgbimage.width ) + " ";
    String sh = Integer.toString ( rgbimage.height ) + "\n";
    byte [] bytes = new byte[3*isize];
    for ( int i = 0, k = 0; i < isize; i++, k+=3 ){
        bytes[k] = (byte) (rgbimage.ibuf[i].R);
        bytes[k+1] = (byte) (rgbimage.ibuf[i].G);
        bytes[k+2] = (byte) (rgbimage.ibuf[i].B);
     }
     try {
        File f = new File ( args[2] );
        OutputStream o = new FileOutputStream( f );
DataOutputStream out = new DataOutputStream ( o );
        out.write ( P6 );
        out.writeBytes ( sw );
        out.writeBytes ( sh );
        out.write ( colorLevels );
        out.write ( bytes );   //save 8-bit RGB data
        out.close();
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }
    System.out.printf("Decoded data saved in %s \n", args[2] );
  }
}
```

You can now put all the java files, **common.java, Decode.java, Encode.java, RgbYcc.java,** and **Test_encode_ppm.java** in the same directory and compile them with the command,

```
$javac *.java
```

The main class **Test_encode_ppm.class** is generated. You have to supply three filenames to do the test, the first being the input PPM file, the second for storing YCbCr data, and the third for saving the decoded RGB data. For example you can issue a command like the following to carry out the test:

```
$java Test_encode_ppm ../data/beach.ppm t.ycc t.ppm
```

When it is executed, it does the following:

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

You can view the PPM files using "xview"; the command "xview ../data/beach.ppm" displays the original RGB image and the command "xview t.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 "t.ycc" by a factor of two. If you want to find out the file sizes, you can issue the command "ls -l ../data/beach.ppm t.ycc t.ppm". Upon executing this command, you should see a display similar to the following:

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

The first column indicates the file sizes in bytes. As you can see, files "beach.ppm" and "t.ppm" have identical size but the file "t.ycc" that contains YCbCr data is only half the size of the file "beach.ppm" that contains RGB data.

In our experiment, the original RGB image (beach.ppm) is shown in Figure 5-5a and the restored RGB image (t.ppm) is shown in Figure 5-5b.

Figure 5-5a

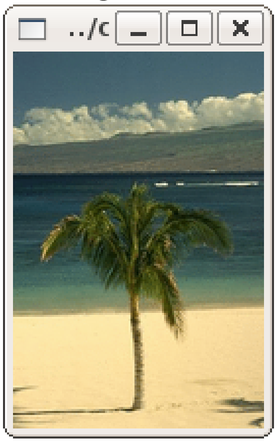

Figure 5-5b