

Chapter 2

Image Representation and Display in Java

- Java 2 : convenient image representation making processing simple. straightforward.
- Solution provided by a standard Java API.

2.1 Our Own Gray Level Image Object

8-bit GL image

```
byte [] [] image = new byte [640][480];  
// each pixel stored in one byte of memory  
// creates storage for an 8 bit, 640×480 pixel image.
```

Better way of determining the storage requirements

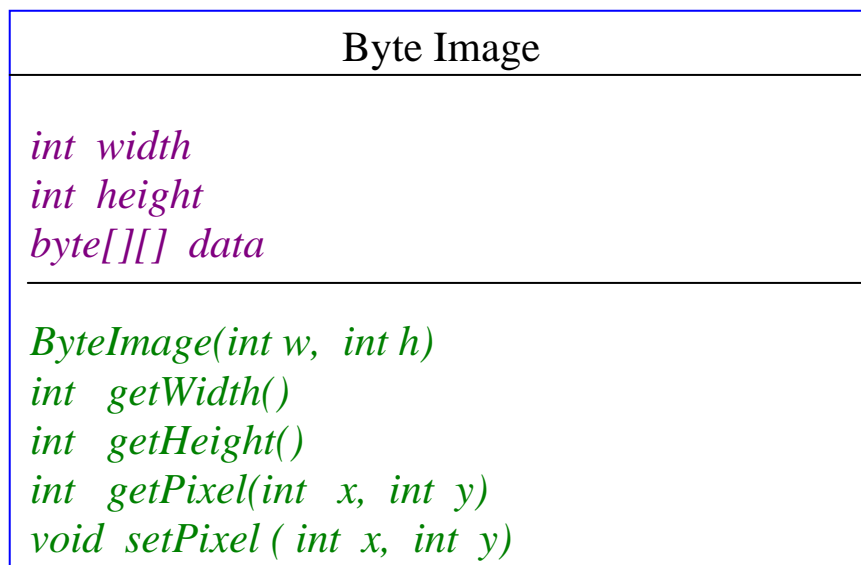
```
int height, width;  
...  
// Determine width and height;  
...  
byte [] [] image = new byte [height][width];
```

Note: three variables define an image:

- 2 integers for dimensions
- a 2D array of bytes to hold gray level data.

A more structured program

- define an *image class* that encapsulates these three variables
- create *an instance* of this class. Furthermore, we want to
- *hide information* so that dimensions and pixel values be *private* to image object but be able to access these variables by *invoking public methods* that form the *interface* to the class.
- This separation of *interface* from *implementation* allows the programmer to think in terms of the *behavior of an image object* rather than details of how the values are stored.
- design in the UML notation:



A class for 8-bit gray level images

Note:

- values of byte in Java are –128 to 127, but we need 0 to 255, *getPixel* method must performs this conversion.
- The constructor of the class which is responsible for creating *ByteImage* object, takes arguments w and h

In a client program an image object is created as

```
ByteImage image = new ByteImage(128, 128);
```

Checking: A method should check its parameters to be in the range, e.g. data values 0 to 255, and $w, h > 0$. Java will take care of these, and throws an exception, i.e.

NegativeArraySizeException

ArrayIndexOutOfBoundsException.

The *ByteImage* class in the above UML uses a 2D array, e.g. the pixel (x, y) has its value stored in *data[y][x]*. We can change this to 1D array

```
private byte[] data;
```

A pixel at (x, y) now has value *data[y*width+x]*.

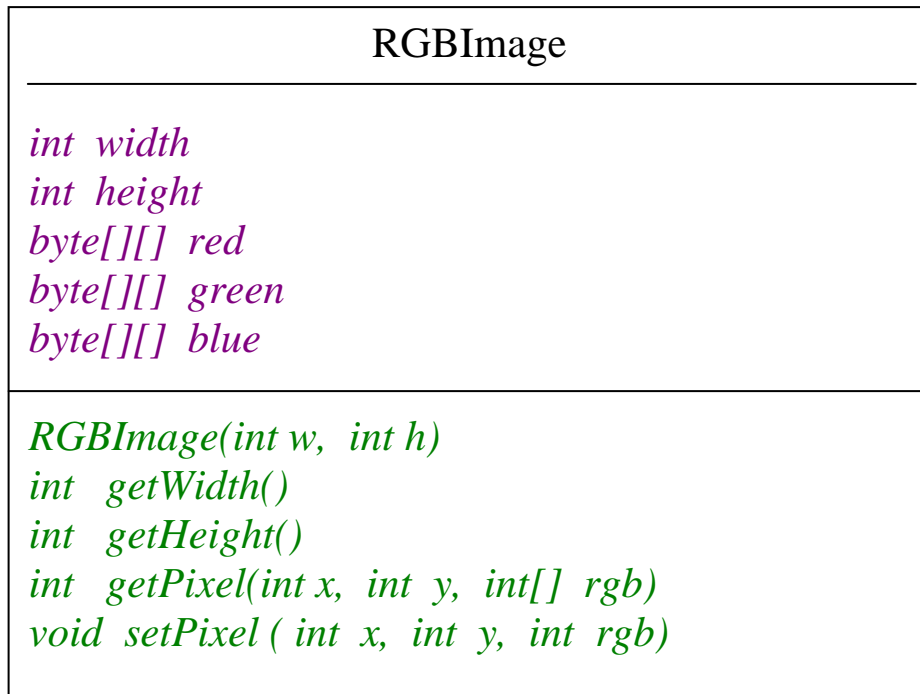
Note : In this case the implementation of *getPixel* and *setPixel* would have to be changed.

Advantages of 1D representation;

- copying an image in memory,
- reading it from a stream, or writing it to a stream is easier

2.2 Creating Color Images

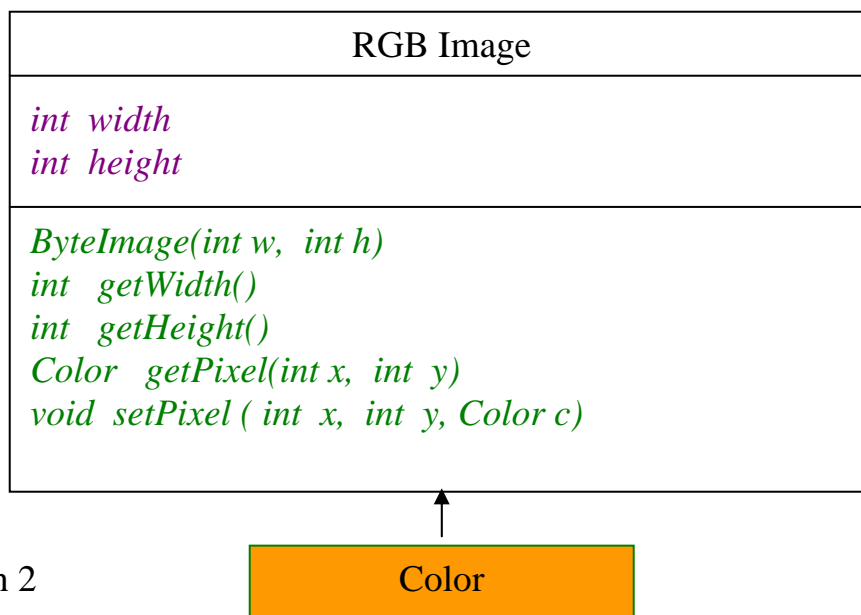
RGB model with 8 bit used for each planer (color), designed as



Design 1

Alternatively:

- assume image consists of a 2D array of data, each array element being a triplet of R, G and B values.
- devise another class to represent the RGB triplet, or use Java Color class from java.awt package.



Design 2

Comparison

- Design 1 is much easier to manipulate the color components independently, but accessing the color of a single pixel requires three array access operations.
- Design 2, pixel color can be retrieved by indexing an array only once, but separate manipulation of the different colors is more difficult.

2.3 Images in Java

- Image manipulation in earlier versions of Java (e.g. Java 1.1) was done via the Image class, which is part of java.awt package.
- To load an imagefile into a program the application can obtain a java.awt.Toolkit object and call its *getImage()* method as

Image image = Toolkit.getDefaultToolkit().getImage(imagefile);

- *getImage()* understands the GIF and JPEG images.
- Toolkit also has a version of *getImage()* that loads an image from a URL.
- A call to *getImage()* returns immediately – it sets up image loading but does not load image data.
- Actual image retrieval is initiated when a call is made to a method (e.g. *drawImage()* to display) that requires the image data.
- Before image data can be used, we must ensure that image loading has been completed. This needs a rather elaborate programming to check the availability of data.

Java 2D Application Programming Interface (API) resolves these types of issues. It has many capabilities for image processing.

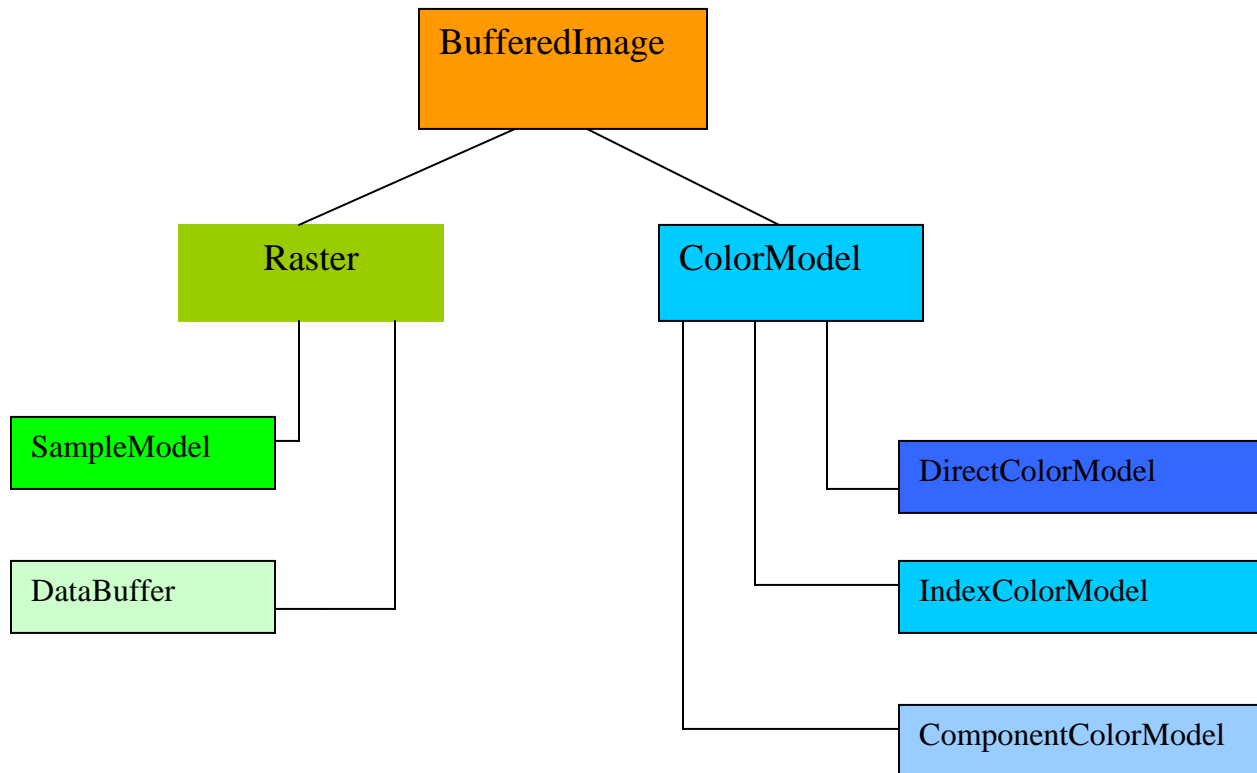
The main classes in Java 2D for image representation are

- *BufferedImage*,
- *ColorModel*,
- *Raster*,
- *SampleModel*
- *DataBuffer*
- Subclasses of above classes

2.4 The *BufferedImage* Class

- Supports a mode of operation in which pixel values are known to be in memory.
- Allowing operations to be performed immediately without the need to wait for the delivery of image data that were needed in earlier versions of Java.
- With this mode of operation, it is possible for an image class to have methods that access individual pixel values, making image processing straightforward.
- *BufferedImage* is a subclass of *Image*, and can be substituted for *Image*, if needed.

BufferedImage consists of a *Raster* and a *ColorModel*,



Composition of *BufferedImage* Object

RasterImage: holds data, and is decomposed as follows.

SampleModel : Every pixel in *Raster* has one or more samples associated with it:

- A binary image has a 1-bit sample for each pixel with 8 samples packed into a byte
- A gray level image has 8-bit sample for each pixel stored in one byte
- A color image has an 8-bit sample for each color store that is in one byte.
- *SampleModel* specifies how the array elements managed by *DataBuffer* are translated into samples of a particular pixel.

DataBuffer : A wrapper for the array(s) used to store pixel data. (Note: a wrapper packages elementary data into classes, and can provide conversion methods for converting from or to other data types).

ColorModel: Specifies how the samples are interpreted. The number of color components specified by the ColorModel must match the number of samples per pixel

IndexColorModel: is used to specify a gray scale.

DirectColorModel: is used to specify values of red, green and blue.

ComponentColorModel: supports representation in which each pixel's samples correspond directly to the component of the color model.

The constructor of BufferedImage takes image **type** and image **dimensions** as its parameters.

Image type is specified by an integer constant defined in BufferedImage class. There are 13 standard types representing different combinations of ColorModel and SampleModel a few of which are given in the Table 2.1 below. The most useful types are TYPE_BYTE_GRAY (for gray level) and TYPE_3BYTE_BGR (for color images). Both use ComponentColorModel, i.e. one component in case of gray level, and three in the case of color images.

Constant	Description
<code>TYPE_BYTE_BINARY</code>	1-bit sample per pixel; 8 samples packed into a byte.
<code>TYPE_BYTE_GRAY</code>	8-bit sample per pixel stored in one byte
<code>TYPE_USHORT_GRAY</code>	16-bit sample per pixel store in a short
<code>TYPE_3BYTE_BGR</code>	8-bit blue, green and red samples, each stored in one byte
<code>TYPE_INT_RGB</code>	8-bit red, green and blue samples, packed into an int.

A few types of *BufferedImage*.

The dimensions and type, of an existing *BufferedImage* object can be obtained using:

- `getWidth()`
- `getHeight()`
- `getType()`

Its color model and pixel values are obtained as

- `getColorModel()`
- `getRGB()`
- `setRGB`

`int getRGB(int x, int y)`

`int getRGB(int x, int y, int w, int h, int[] data)`

`void setRGB(int x, int y, int value)`

`void setRGB(int x, int y, int w, int h, int[] data)`

- *getRGB()* and *setRGB()* represents each pixel value as a 32-bit integer as follows

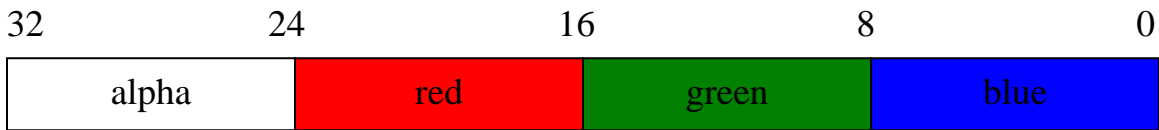


Fig. 2.5 RGB representation

- alpha is the pixel transparency, alpha = 0 → fully transparent pixel, invisible when displayed.
- alpha = 255 → fully opaque, hides whatever is behind it when displayed. Intermediate values indicate some degree of blending of pixel gray level or color with that of the background.

This method of packing color information into a single int, can be inefficient when used in conjunction with `TYPE_3BYTE_BGR` which stores red, green and blue in separate array elements. The *getRGB()* method gives us these three components in the form of one integer, and to process the different colors, we must extract red, green and blue values from this integer. After processing the components they must be packed again into an integer and pass it to *setRGB()* which extract the color components and stores them into arrays used for data storage.

Complications can arise if *getRGB()* and *setRGB* are used on grayscale images. This is due to the fact that the three color components are all equal in a gray scale image For example a graylevel 50 has to alpha = 111111 (opaque), R = G = B = 50 (in base 10)= 00110010 (in base 2), and a call to *getRGB()* will return 111111 00110010 00110010 00110010 = - 13487566.

This can be fixed, but a better solution is provided by *Raster* object which manipulates the pixels at a lower level.

2.5 Raster and WritableRaster

- Java2D *Raster* class contains methods that manipulate the samples of a pixel directly, without the misinterpretation that can occur with *ColorModel*.
- *Raster* is a read-only class for inspecting pixel values only, and *writableRaster* changes the pixel values.

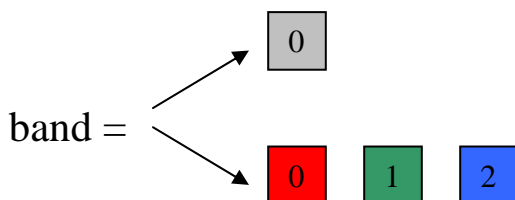
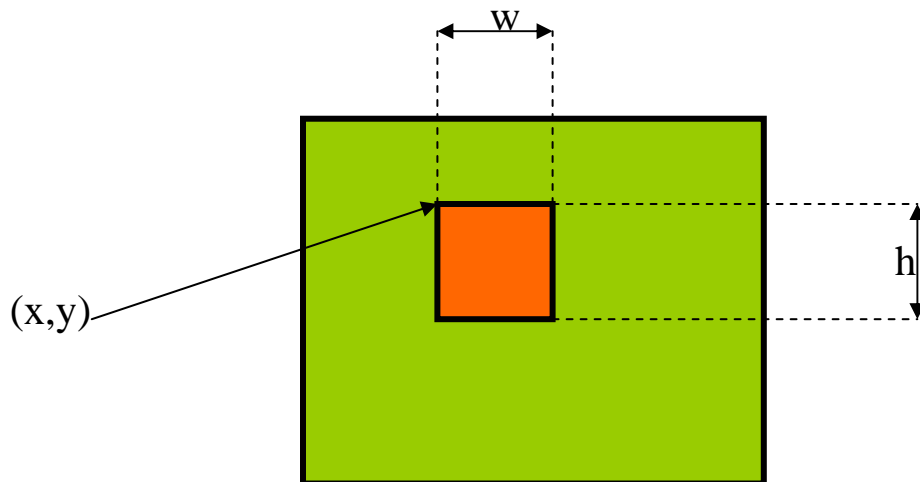
The Raster class methods are

```
int[] getPixel(int x, int y, int[] data)
```

```
int[] getPixels(int x, int y, int w, int h, int[] data)
```

```
int getSample(int x, int y, int band)
```

```
int[] getSamples(int x, int y, int w, int h, int band, int[] data)
```



Note : Each of the above methods also has float and double type versions, where the method and the data are declared either as floats or doubles, e.g.

```
double getPixel(int x, int y, double[] data)
```

The writableRaster methods are

```
void setPixel(int x, int y, int[] data)
```

```
void setPixels(int x, int y, int w, int h, int[] data)
```

```
void setSample(int x, int y, int band)
```

```
void setSamples(int x, int y, int w, int h, int band, int[] data)
```

Note : Each of the above methods also has float and double data type versions.

Example: Suppose we have a grayscale *BufferedImage* called *picture*, and we wish to divide all its gray levels by 2.

```
writableRaster raster = picture.getRaster();
int value;
for (int y = 0; y < picture.getHeight(); ++y)
    for (int x = 0; x < picture.getWidth(); ++x)
    {
        value = raster.getSample(x, y, 0)/2;
        raster.setSample(x,y, 0, value);
    }
```

JPEG Images

- The reading from and writing of JPEG images to data streams are done through the package `com.sun.image.codec.jpg`.

- The main class for these tasks is `JPEGCodec`, a *factory class* with static methods that manufactures instances of

JPEGImageDecoder (for reading images) and *JPEGImageEncoder* (for writing them).

- To load an image from the file `picture.jpg`:

```
FileInputStream fileStream = new FileInputStream ("picture.jpg");
JPEGImageDecoder input = JPEGCodec.createJPEGDecoder(fileStream);
BufferedImage image = input.decodeAsBufferedImage();
```

- To write the image to a new JPEG file:

```
FileOutputStream fileStream = new FileOutputStream ("out.jpg");
JPEGImageEncoder output = JPEGCodec.createJPEGEncoder(fileStream);
output.encode(image);
```

The Java Advanced Imaging API, or JAI:

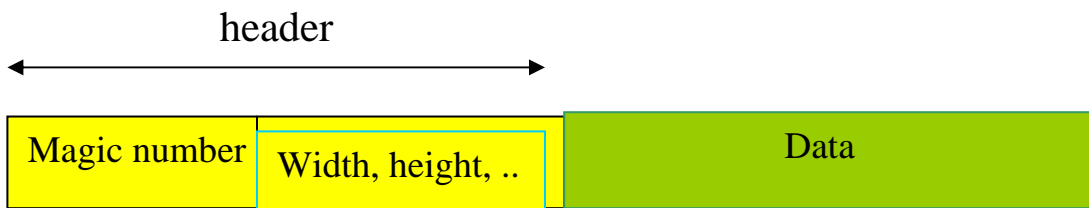
- is an extension to Java, but is not a standard part of Java 2.
- An image processing task is performed by linking together various operators in a processing graph.
- This graph is like a map that specifies the paths along which image data can flow from one operator to another.
- JAI is a significant enhancement to Java's image processing capabilities.

Note : The goal of this course is to explore basics of image processing and computer vision, and how can can be implemenetd using an object oriented language such as Java. The relatively simple model for image processing provided by classes such as *BufferedImage* and *BufferedImageOp* (to be discussed later), enable us to achieve this goal without programming issues obscuring our main goal.

2.6 Image File Formats

Simple format:

- Header – image height, width, and a signature or “magic number” a sequence of bytes designed to identify the image format.
- Data part



File formats generally fall into three categories:

Device-spezilized formats: designed for specific hardware, and suffer from lack of portability.

Software specialized formats: designed to be used with a particular program

- PCX and Windows bitmap formats found on PCs
- MacPaint used on Macintosh computers.

Intechange formats: designed to

- facilitate the exchange of image data between users
- usable with different hardware and software.

Image compression is standard features of interchange formats, some of which are:

- **GIF** (Graphic Interchange Format): came about for images on WWW.
- **JFIF** (JPEG File Intechnage Format): also appeared with WWW, uses special compression called JPEG to be discussed later.
- **TIFF** (Tagged Image File Format): Does not require information to be stored in fixed locations in an image file, instaed it uses special strings or codes identify a particular data items.
- **PNG** (Portable Network Graphics): is a replacement for GIP for legal reasons.
- **PGM** (Portable Gray Format): a popular format for grayscale image on Unix systems.
- **FITS** (Flexible Image Transport Format): allows large amount of information to be placed in the header.

The Potable Formats Family

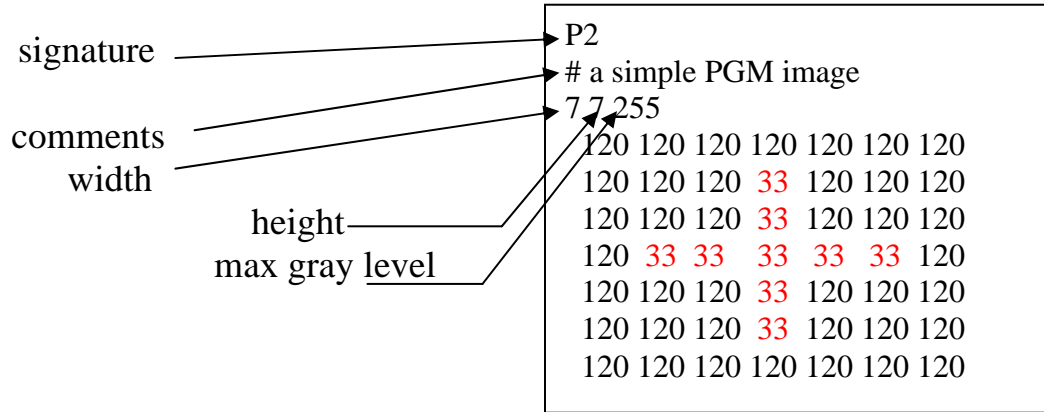
This family consists of PGM (Portable Grayscale Format), PBM (Portable Bitmap), PPM (Portable Pixmap). These formats are identified by two character signature, e.g. P1, P2

Signature	Image type	Storage type
P1	binary	ASCII
P2	grayscale	ASCII
P3	RGB	ASCII
P4	binary	raw byte
P5	grayscale	raw byte
P6	RGB	raw byte

Signature of various portable formats

Fig shows a very simple 7 by 7 grayscale image representation as ASCII and raw PGM files (the raw format may have unprintable characters).

The advantage of ASCII format is that pixel values can be easily examined and changed using a text editor.



However, the raw format is much more compact, and takes about one fourth space of the ASCII format.

```

P5
# a simple PGM file
7 7 255
xxxxxxxxxx ! xxxxxxx ! xxxx !!!!! xxxx ! xxxxxxx ! xxxxxxxxxxxx
  
```

The PNG format

- one of the newer formats.
- supports grayscale up to 16 bits/pixel, and RGB with up to 16 bits per band, or 48 bits/pixel.
- PNG also supports transparency value
- The data stream is compressed, and is lossless

A PNG file consists of an 8 byte signature, followed by a series of “chunks”.

Each chunk consists of

- a 32 bit integer giving number of byte in the chunk’s data field
- a four byte code to indicate chunk type
- a 32 bit cyclic redundancy check (CRC) for the chunk to test data validity.

Table 2.3 below shows examples of chunk type.

Chunk type	Usage
IHDR	Image header
IDAT	Image data
IEND	End of image file
PLTE	Color palette
gAMA	Gamma correction
pHYs	Pixel physical dimensions
teXt	Textual comment
tIME	Time of last modification

Table 2.3 Chunk types.

Every PNG file must have a chunk type IHDR (header), one or more IDAT (data), and must end with an IEND chunk.

2.7 Display of Images

Hardware

Images are viewed on a cathod-ray tube (CRT) monitor.



- There are three electron guns emitting narrow beams of electrons which are swept across the front face of the tube by deflection coils.
- The face is coated with a pattern of dots of three types of phosphor which emit varying amount of red, green and blue light when struck by varying numbers of electrons.
- The phosphor dots are typically arranged into triangular groups of three, one for each color. They are sufficiently small to be unresolvable by the human eye, and we see each point on the screen as a mixture light from the RGB phosphors of light. The intensities of the light beams determine the value of values of R, G and B.

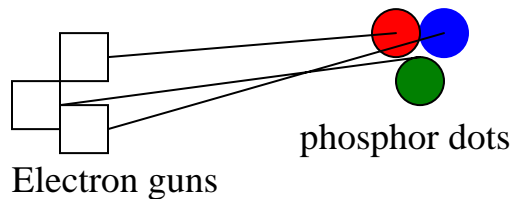


Fig 2.7 CRT principles

- The computer monitor also has three independent electron guns for the R,G, B components of a video signal, but this does not guarantee that we can display 24-bit RGB images.
- The limiting factor is the graphic hardware of the computer graphic card that has a fixed quantity of on-board memory that can be used for different spatial and color resolutions. If spatial resolution is increased so that large images can be displayed on screen, then color resolution may need to be reduced.

Example: A graphic card of 2 Mbyte of video RAM can display

$$(24 \text{ bit color}) \times (\text{resolution of } 800 \times 600) = 1,440,000 \text{ bytes}$$

If the spatial resolution is increased to 1024×768

$$(24 \text{ bit color}) \times (\text{resolution of } 1024 \times 768) = 2,359,296 \text{ bytes}$$

- exceeding the limits of the hardware. In this case either the the whole image cannot be displayed, or the number of color has to be reduced say to 16 bits instead of 24 bits.

Image Display using Java

- An image can be drawn using methods of *Graphics* or *Graphics2D* classes.
- To display an image using AWT, we must extend an existing AWT component and override its *paint()* method of *Canvas* class.
- The *Canvas* class is ideal for image display because it has a blank area on which nothing is drawn by default.
- Instances of the new class, called *ImageCanvas* can be included with other GUI components to create interactive image processing applets or applications.
- The code below shows how *ImageCanvas* can be implemented.

```
import java.awt.*;
```

```
public class ImageCanvas extends Canvas
{
    BufferedImage image;
    public ImageCanvas ( bufferedImage img)
    {
        image = img;
    }
}
```

```
public void paint(Graphics grph)
{
    grph.drawImage(image, 0,0, this);
}
}
```

- If a window was resized , *paint()* method of an *ImageCanvas* is called automatically to refresh the display.
- The *Graphics* object that *paint()* receives is the graphics context for the *ImageCanvas*, and we must call one of the *drawImage()* methods provided by this graphic context to ensure that the image is painted onto the canvas whenever necessary.

- The prototype for *drawImage* is

```
public boolean drawImage(Image img, int x, int y, ImageObserver obs)
```

In the above *ImageObserver* is an object that will notify of changes in the status of the image as it is loaded. In the above listing, *this* is a reference to *ImageCanvas* itself. The *drawImage()* given above is one of the six versions that are provided by a *Graphics* object. In another version, *drawImage()*, in addition to *x*, and *y* has two other parameters *w* and *h* that specify the width and height of the image display, so that the image can be magnified or shrunk as necessary.

We can use Swing GUI components, which offer improved replacement for AWT, and have many new and sophisticated classes. However, AWT-based components must not be mixed with Swing components, since the former is platform dependent (e.g. Microsoft Windows), whereas Swing is purely in Java. Therefore, we should use Swing components only in GUI-based programs written for Java 2 environment.

The Swing provides *JLabel* instead of *Canvas*, which is an object used to add a short piece of text, an icon or a combination of the two to a user interface. A *JLabel* can be constructed from an *ImageIcon* object, which in turn can be constructed from an *Image* or *BufferedImage* object. If there is a *BufferedImage* called *image*, then a component to display this image can be created as follows:

```
ImageIcon icon = new ImageIcon(image);  
JLabel view = new JLabel(icon);
```

It is possible to create a new component that is dedicated to the task of image display. We can do this by extending *JLabel*. A possible implementation is shown below. One important difference between this class and *ImageCanvas* discussed before is that we must override the *paintComponent()* method, rather than *paint()*. The other feature of the class – instance variable *viewSize* and the methods after *paintComponent()* – are there to support scrolling of the component when it is embedded in a *JScrollPane*.

```
import java.awt.*;  
import java.awt.image.*
```

```

import javax.swing.*;

public class ImageView extends JLabel implements Scrollable
{
    private BufferedImage image; //image to be displayed
    private Dimension viewSize; // size of view
    public ImageView(BufferedImage img){
        image = img;
        int width = Math.min(256, image.getWidth());
        int height = Math.min(256, image.getHeight());
        viewSize = new Dimension(width, height); //set max viewSize to 256 by 256
        setPreferredSize(new Dimension(image.getWidth(), image.getHeight()));
        // set preferred size to image dimension, if not more than 256
    }

    public void paintComponent(Graphics g) {
        g.drawImage(image, 0, 0, this);    }

    public void setViewSize(Dimension newSize) {
        viewSize.setSize(newSize);    }

    public Dimension getPreferredSizeScrollableViewportSize(){
        return viewSize;    }

    public int getScrollableUnitIncrement(Rectangle rect, int orient, int dir) {
        return 1;    }

    public int getScrollableBlockIncrement(Rectangle rect, int orient, int dir) {
        if (orient == SwingConstants.HORIZONTAL)
            return image.getWidth() / 10;
        else
            return image.getHeight() / 10;    }

    public Boolean getScrollableTracksViewportWidth() {
        return false;    }

    public Boolean getScrollableTracksViewportHeight() {
        return false;    }
}

```

}

The constructor of *ImageView* sets its preferredsize to image dimensions, so that the entire image will be visible if the component is used on its own. The *viewSize* is set to a maximum 256 by 256. When an *ImageView* is embedded in a *JScrollPane*, the *viewSize* variable represents the dimensions of the view port through which we see the image.

An Image Viewer Application: *ImageView* class can be used for a complete image viewing application. The program in the CD loads an image from a file and displays it using an *ImageView* component. The application listens for mouse event occurring within the viewing area, translates cursor coordinates into image coordinates, and retrieves pixel gray level or color at those coordinates and displays these data in a panel below the image. The program has a magnifying glass allowing a small window to pop up containing a magnified view of the region surrounding the cursor.

2.8 Printing of Images

- Most printers produce a binary output, i.e. they either print a black dot or blank (white).
- Newspaper photographs simulate a grayscale by printing tiny black dots of varying size. The human visual system has a tendency to average brightness over small areas so that black dots and white background merge and are perceived as shades of gray.
- This process is called **halftoning**, and several methods of implementing it are available, as follows.

Patterning: In this method each pixel is replaced by a pattern taken from a binary font. An example of this font made up of 3 x 3 matrix of pixels is shown below for representing 10 graylevel values, from 0 to 9. In this case the width and height of the image is increased by a factor of 3.

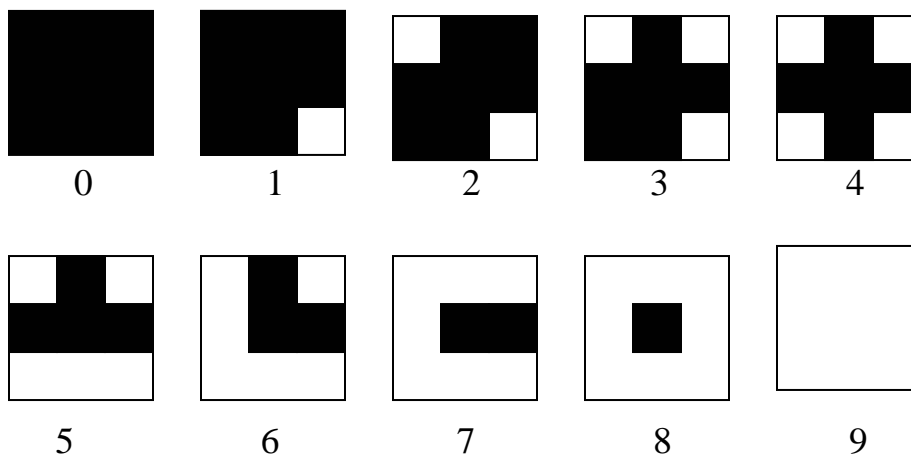


Fig. 2.8 Halftoning via patterning

Dithering: This is accomplished by **thresholding** the image against a dither matrix. These matrices are square with dimension of power of 2. The elements of the matrix are thresholds. The first two dither matrices for an 8 bit grayscale image are

$$D_1 = \begin{bmatrix} 0 & 128 \\ 192 & 64 \end{bmatrix}, \quad D_2 = \begin{bmatrix} 0 & 128 & 32 & 160 \\ 192 & 64 & 224 & 96 \\ 48 & 176 & 16 & 144 \\ 240 & 112 & 208 & 80 \end{bmatrix}$$

-- The matrix is laid on the image. The output pixel value becomes white (255) if the input pixel values is greater than the matrix element, or becomes black (0) if it is less than the matrix element.

-- For seeng better details, the image is enlarged by a factor of 2^m where m is the dither matrix size.

Error Diffusion : We start selecting a threshold, typically 128 for an 8-bit image. If $f(x,y) > 128$, the pixel at (x,y) in the output is set to white otherwise it is set to black. For pixels that are close to 128, the error will be large, and in order to reduce the effect, this error is spread or diffused to the neighboring pixels. A method for diffusing is shown below, where the error is spread among 4 neighbors that are ahead of the pixel, assuming top to bottom and left to right traversal. The values shown in Fig. are the factors by which the error is multiplied, the result is added to the corresponding pixel value

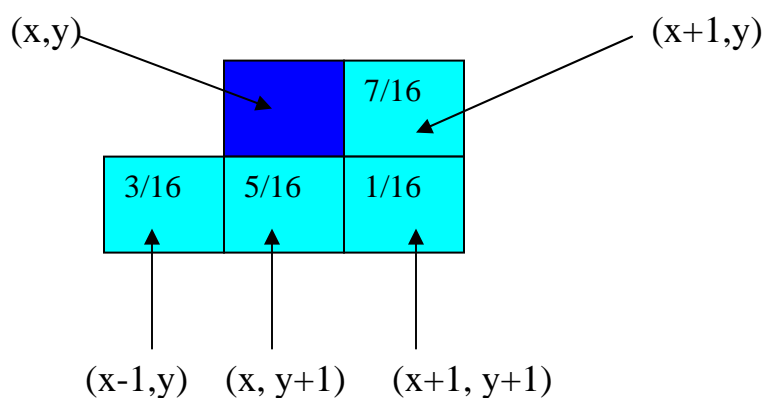


Fig 2.9 Error diffusion factors.

The algorithm by Floyd-Steinberg for error diffusion is

```
threshold = (black + white)/2
for ( all x and y ) {
    if (f(x,y) < threshold)    {
        g(x,y) = black // g(x,y) is the output image
        error = f(x,y) - black    }
    else {
        g(x,y) = white
        error = f(x,y) - white
    }
    f(x+1,y) = f(x+1, y) + 7*error/16
    f(x-1, y+1) = f(x-1, y+1) + 3*error/16
    f(x, y+1) = f(x, y+1) + 5*error/16
    f(x+1,y+1) = f(x+1, y+1) + error/16
}
```

The Java implementation is shown at the end of this chapter.

- The CMYK color model is used for printing of color images. The color printer has four inks – cyan, magenta, yellow and black. The output produced is binary, i.e. either one of the four colors is printed or absent of any point. One of the methods of halftoning is used for each color.

// Java implementation of error diffusion for graylevel images.

```
import java.awt.image.*;
import java.io.IOException;
import javax.swing.JFrame;
import com.pearsoneduc.ip.gui.*;
import com.pearsoneduc.ip.io.ImageDecoderException;
import com.pearsoneduc.ip.op.OperationException;

public class ErrorDiffusion extends OperationViewer {

    public ErrorDiffusion(String imageFile)
        throws IOException, ImageDecoderException, OperationException {
```

```

    super(imageFile);
    setTitle("ErrorDiffusion: " + imageFile);
}

// Check that we have a greyscale image

public boolean imageOK() {
    return inputImage.getType() == BufferedImage.TYPE_BYTE_GRAY;
}

// Perform Floyd-Steinberg error diffusion

public void processImage() {

    // Create a binary image for the results of processing

    int w = inputImage.getWidth()-1;
    int h = inputImage.getHeight()-1;
    outputImage = new BufferedImage(w, h, BufferedImage.TYPE_BYTE_BINARY);

    // Work on a copy of input image because it is modified by diffusion

    WritableRaster input = inputImage.copyData(null);
    WritableRaster output = outputImage.getRaster();
    final int threshold = 128;
    int value, error;

    for (int y = 0; y < h; ++y)
        for (int x = 0; x < w; ++x) {

            value = input.getSample(x, y, 0);

            // Threshold value and compute error

            if (value < threshold) {
                output.setSample(x, y, 0, 0);
                error = value;
            }
            else {
                output.setSample(x, y, 0, 1);
                error = value - 255;
            }

            // Spread error amongst neighbouring pixels

            value = input.getSample(x+1, y, 0);
            input.setSample(x+1, y, 0, clamp(value + 0.4375f * error));
            value = input.getSample(x-1, y+1, 0);
            input.setSample(x-1, y+1, 0, clamp(value + 0.1875f * error));
        }
    }

```

```

        value = input.getSample(x, y+1, 0);
        input.setSample(x, y+1, 0, clamp(value + 0.3125f * error));
        value = input.getSample(x+1, y+1, 0);
        input.setSample(x+1, y+1, 0, clamp(value + 0.0625f * error));
    }

}

// Forces a value to a 0-255 integer range

public static int clamp(float value) {
    return Math.min(Math.max(Math.round(value), 0), 255);
}

public static void main(String[] argv) {
    if (argv.length > 0) {
        try {
            JFrame frame = new ErrorDiffusion(argv[0]);
            frame.pack();
            frame.setVisible(true);
        }
        catch (Exception e) {
            System.err.println(e);
            System.exit(1);
        }
    }
    else {
        System.err.println("usage: java ErrorDiffusion <imagefile>");
        System.exit(1);
    }
}

```


A simple Swing application to load an image from a file and display it is shown below.

```
import java.awt.*;
import java.awt.image.*
import javax.swing.*;
import com.pearsoneduc.ip.io.*;

public class Display extends JFrame
{
    public Display(String filename)
    {
        super(filename);
        ImageDecoder input = ImageFile.createImageDecoder(filename);
        BufferedImage image = input.decodeAsBufferedImage();
        JLabel view = new JLabel(new ImageIcon(image));
        getContentPane().add(view);
        addWindowListener(new WindowAdapter())
```

```

        {
            public void windowClosing(WindowEvent event){
                System.exit(0);}
        });
    }

public static void main(String[] argv)
    {
        if (argv.length > 0)
        {
            try{
                JFrame frame = new Display(argv[0]);
                frame.pack();
                frame.setVisible(true);
            }
            catch (Exception e){
                System.err.println(e);
                System.exit(1);}
        }
        else{
            System.err.println("usage: java Display imagefile");
            System.exit(1);
        }
    }
}

```

The Swing's `JFrame` class is extended so that we have a window in which to display the image. The line `super(filename);` invokes the parent class constructor with image filename as a parameter so that the filename is placed in the titlebar of the window. The next two lines, i.e. those starting with `ImageDecoder` and `BufferedImage`, load the image from the file. The next line creates a `JLabel` in which the image is drawn, and this is added to the frame's content pane with `getContentPane().add(view)`. The next three lines set up the event handling that will allow us to remove the frame by clicking on the appropriate button on the titlebar. Finally, the three lines starting with `JFrame` create the frame, resize it to match the size of its contents and then make it visible.