Bitwise Logical Operators and Image Processing Convolution Filters

Class 9 – Prepared by Nicolas Bergeron

Outline

- Logical Bitwise Operators
  - & : Bitwise AND
  - | : Bitwise OR
  - ^ : Bitwise Exclusive OR (XOR)
  - ~ : Bitwise Complement
  - <<: Binary Left Shift Operator
  - >>: Binary Right Shift Operator
- Applications
  - Setting flags and masks
  - Encoding 4-channel colors into unsigned int
  - Multiply by powers of 2, or divide by powers of 2 using shift operators
  - SWAP values in integers using XOR
- Intro to Image Processing
  - Convert to Grey Scales
  - Gamma correction
  - Convolution Filters

Logical Bitwise Operators

Outline

- Logical Bitwise Operators
  - & : Bitwise AND
  - | : Bitwise OR
  - ^ : Bitwise Exclusive OR (XOR)
  - ~ : Bitwise Complement
  - <<: Binary Left Shift Operator
  - >>: Binary Right Shift Operator
- Applications
  - Setting flags and masks
  - Encoding 4-channel colors into unsigned int
  - Multiply by powers of 2, or divide by powers of 2 using shift operators
  - SWAP values in integers using XOR
- Intro to Image Processing
  - Convert to Grey Scales
  - Gamma correction

Bitwise Operators

- Bitwise operators are available with most programming languages, they are even available in Assembly Language
- These operators can be applied on all integer types (char, byte, short, int, long)
- Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows:
    - a == 0011 1100
    - b == 0000 1101

& : Bitwise AND

- Binary AND Operator copies a bit to the result if it exists in **both** operands. Example:
  - `a == 60 == 0011 1100`
  - `b == 13 == 0000 1101`

  - `a & b   == 0000 1100 == 12`

| : Bitwise OR

- Binary OR Operator copies a bit to the result if it exists in **<u>either</u>** operands. Example:
  - `a == 60 == 0011 1100`
  - `b == 13 == 0000 1101`

  - `a | b   == 0011 1101 == 61`

^ : Bitwise Exclusive OR (XOR)

- Binary XOR Operator copies the bit if it is set <u>**in one operand but not both**</u>.
  - `a == 60 == 0011 1100`
  - `b == 13 == 0000 1101`

  - `a ^ b ==   0011 0001 == 49`

~ : Bitwise Complement

- Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
  - `a == 60 == 0011 1100`

  - `~a       == 1100 0011`

<<: Binary Left Shift Operator

- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.

  - `a == 60 == 0011 1100`

  - `a << 2  == 1111 0000 == 240` (Notice this is also 60*4)

>>: Binary Right Shift Operator

- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

  - `a = 60 = 0011 1100`
  - `a >> 2 = 0000 1111 = 15   (Notice this is also 60/4)`

Examples in Java

```java
public class Test {

   public static void main(String args[]) {
      int a = 60; /* 60 = 0011 1100 */
      int b = 13; /* 13 = 0000 1101 */
      int c = 0;

      c = a & b;        /* 12 = 0000 1100 */
      System.out.println("a & b = " + c );

      c = a | b;        /* 61 = 0011 1101 */
      System.out.println("a | b = " + c );

      c = a ^ b;        /* 49 = 0011 0001 */
      System.out.println("a ^ b = " + c );

      c = a << 2;       /* 240 = 1111 0000 */
      System.out.println("a << 2 = " + c );

      c = a >> 2;       /* 15 = 1111 */
      System.out.println("a >> 2  = " + c );

   }
}
```

Logical Bitwise Operators Applications

- There are many opportunities to use logical bitwise operators in a program.
  - Dealing with binary data such as Colors in 32-bit int types
    - For example: integers encoding 4-color channels ARGB 8-bit per channel
  - Multiply and Divide by powers of 2 using shift operators
  - SWAP values of 2 integers using XOR (this is kind of a hack)

Application : Encoding Colors in a single “int”

ALPHA								RED								GREEN								BLUE							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

```java
// This is the Color class in the JDK
public final class Color extends Paint {
    private final int argb;
    private final float r;
    private final float g;
    private final float b;
    private final float a;

    public Color(float r, float g, float b, float a) {
        super(Type.COLOR, false, false);
        int ia = (int)(255.0 * a);
        int ir = (int)(255.0 * r * a);
        int ig = (int)(255.0 * g * a);
        int ib = (int)(255.0 * b * a);
        this.argb = (ia << 24) |(ir << 16) | (ig << 8) | (ib << 0);
        this.r = r;
        this.g = g;
        this.b = b;
        this.a = a;
    }


    // ...
}
```

Bit Shifting: Multiply or Divide by powers of 2

- Shift operators have this property of multiplying by powers of 2 (2, 4, 8, 16, 32, 64, …)
- Examples:

$$5 \ll 2 == 5 * 2^2 == 20$$
$$20 \gg 2 == 20 / 2^2 == 5$$

$$13 \ll 4 == 13 * 2^4 == 208$$
$$208 \gg 4 == 208 / 2^4 == 13$$

Image Processing

Images Operations in Java

- Reading Image
  - BufferedImage img = ImageIO.read(new File(filename));
- Iterating over the image pixels

```
for (int j=0; j<img.getHeight(); ++j){
    for (int i=0; i<img.getWidth(); ++i){
        // do something!
    }
}
```

- Reading pixel colors

```
Color color = img.getRGB(i, j);
```

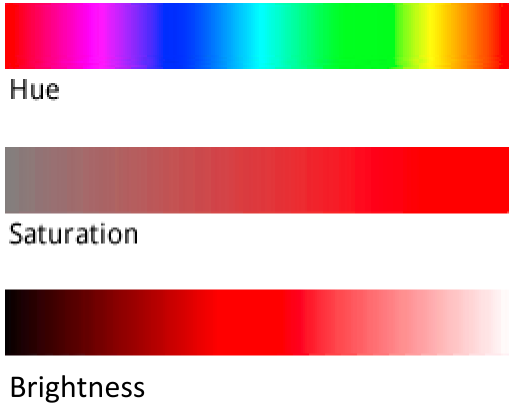
- Setting image pixel color

```
img.setRGB(i, j, color.getRGB());
```

- Saving Modified Image

```
File outputfile = new File(outputFilename);
ImageIO.write(img, "png", outputfile);
```

Color Space Conversion



ALPHA								RED								GREEN								BLUE							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

RGB Color Space

- Color encodes the intensity in Red, Green and Blue
- Java Color natively encodes RGB colors + Opacity Channel (Alpha)
- To Convert to HSB:

  Color.RBGtoHSB() (usage below)

HSB Color Space

- Color encodes the perceptual Hue, Saturation and Brightness
- To set a color from HSB in Java:

  Color.getHSBColor(H, S, B)
- To Convert to RGB:

  Color.HSBtoRGB()

```
float[] hsb = Color.RGBtoHSB(rgb.getRed(), rgb.getGreen(), rgb.getBlue(), null);
float hue        = hsb[0]; // in the range [0,1]
float saturation = hsb[1]; // in the range [0,1]
float brightness = hsb[2]; // in the range [0,1]
```

Direct Color Mapping

- **Black and White:** To convert a colored image to grey scale, we can calculate the average intensity of each channel
  - R = G = B = (R + G + B) / 3;
- To change the brightness of an image, can convert to HSB, increase the B value, and convert back to RGB
- **Gamma Correction**: It is sometimes useful to apply a non-linear mapping of the brightness, such as displaying images on a projector where the colors don’t map the same as on the screen (see example on the right)
  - Convert color to HSB
  - $B_{corrected} = B^{Gamma}$ B must be in the range [0, 1]
  - Convert HS$B_{corrected}$ to RGB

Convolution Filter

- In image processing, a kernel, convolution matrix, or mask is a small matrix. It is useful for blurring, sharpening, embossing, edge detection, and more. This is accomplished by means of convolution between a kernel and an image.
- In simpler terms, it combines the color of the neighboring pixels of the current pixel

Input image

Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Other Convolution Filters

Operation	Kernel	Image result
Edge detection	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

Code provided

- Open the ImageProcessing.zip Netbeans project
- The java file does apply a convolution kernel on an input image
- Notice the following methods:
  - ***ApplyConvolutionFilter***
    - will apply a kernel matrix provided onto the image
  - **ApplyKernel** (used by ApplyConvolutionFilter)
    - for each pixel which calculates the weighted sum of neighboring pixels
    - Wraps indices to avoid going beyond the image boundaries
    - Clamps the final colors to avoid having a value beyond the interval [0, 255]

Exercise 1

- From the code provided, add methods to do the following image processing operations
  - Edge Detection Filter
  - Sharpen Filter
  - Box Blur
- In main method, run all image processing filters and save results in separate files

References

- Java Operators Documentation

  https://www.tutorialspoint.com/java/java_basic_operators.htm

- Wikipedia on Convolution Filters

  https://en.wikipedia.org/wiki/Kernel_(image_processing)