EXERCISE 3:
IMAGE PROCESSING PART 1

Multimedia Database SS 23

Prof. Dr. Harald Kosch/ Prof. Dr. Mario Döller
Kanishka Ghosh Dastidar, Alaa Alhamzeh
(Exercises)

1a) Image Formats

- **Vector Image**
  - Makes use of sequential commands or mathematical statements which place lines or shapes in a 2-D or 3-D environment.
  - Each control point has a definite (x,y) position and determines a graphic primitive, which may be shapes, curves, splines
  - Occupies less space,
  - Suitable for printing, since vector images scale well

- **Raster Image**
  - Raster Images are pixel based
  - Use bitmaps to store information
  - Consumes more memory
  - Scaling up leads to degradation.

1b) Image Formats: GIF

- **Color model:** RGB
- **Color depth:** 8 Bits per pixel – range of 24 Bits color space (256/16 M)
- **Lempel-Ziv-Welch (LZW) compression** – lossless
- Small file size
- Can represent animations, logos, etc..
- but not suitable for photos due to color limitation.

1b) Image Formats: PNG

- **Color model**: RGB/RGBA
- **Color depth:** variable – grey scale (1-16 Bpp) Indexed (1-8 Bpp 24 Bits Color palette), TrueColor (24/48 Bpp)
- **Compression "Deflate“:** Combination of LZSS and Huffman Coding (lossless)

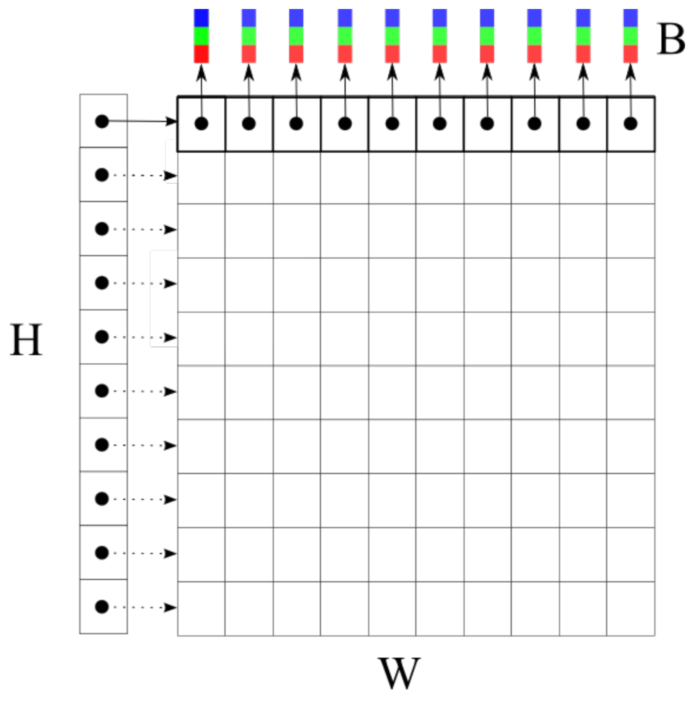
- Doesn‘t support animation

1b) Image Formats: JPEG

- **Color model**: RGB
- **Color depth:** 24 Bits per pixel (most common)
- **DCT-based compression,** lossy
- Doesn‘t support animation, transparency

2a) Image Processing

- Naive internal representation: 3-D Array
- int[][][] raster



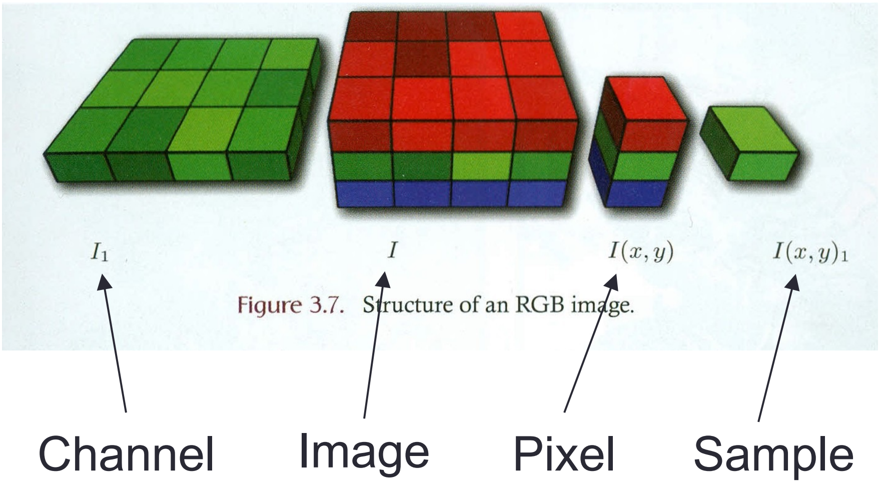


Figure 3.7. Structure of an RGB image.

2) Image Processing

- **24bpp RGB image: 1 Byte = 8 bits per sample**
  - Size(raster) = w x h x b samples (sample = channel value)
  - ⇒ **Efficient implementation would require at most** w x h x b Bytes of memory

- **Practically -> more**
  - Java: multi-dimensional arrays = array of an array

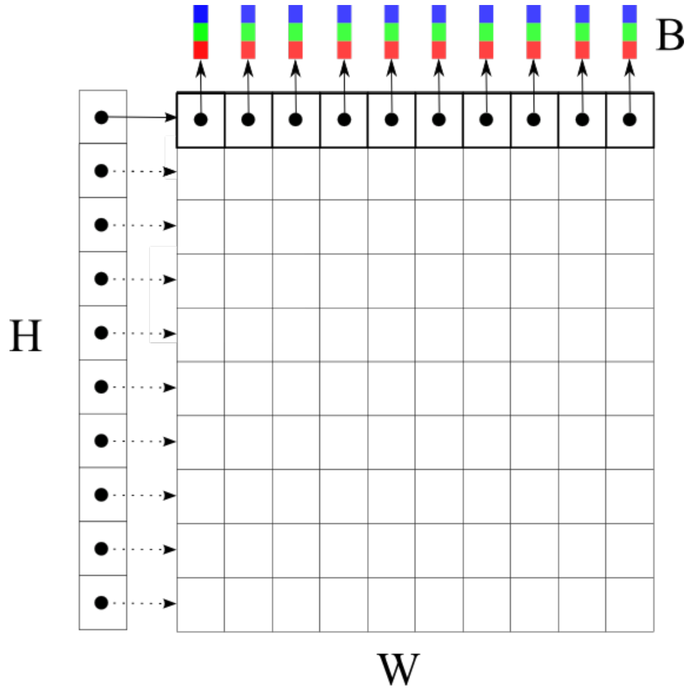
2a) Image Processing

- **Memory Consumption using 3-D raster array**
- **References (Pointers):**
  - H+WH (4 Bytes for each)
- **Object Header (Array length attribute) :**
  - 1+H +WH (4 Bytes each)

•Total amount of unnecessary memory consumed by this 3-D representation:

  - 4(H+WH) + 4(WH + H +1)

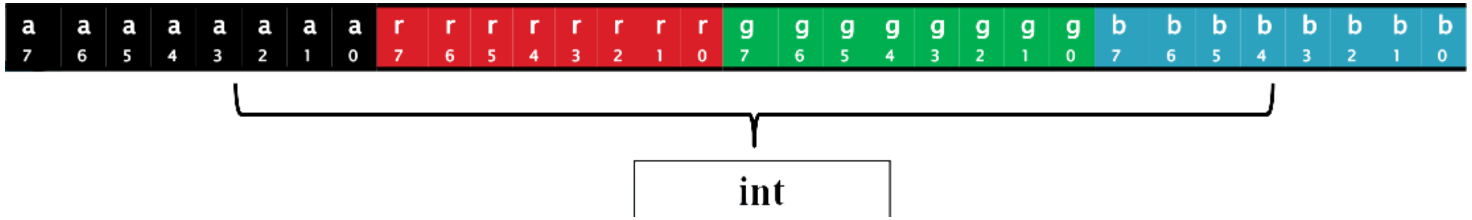
  ~ 8 WH
- **Image data**
  - 3WH
- **Total (W=H=12)**
  - 4(H+WH) + 4(WH + H +1)+4(3WH) = 2980 Bytes



2b) Image Processing

- Till now, a sample was stored in an integer value (**4-Bytes per sample**).

- However, most image processing applications **require a color depth** of at most **24 bits per pixel** (**1-Byte per sample**).
- **Solution:** pack the entire pixel into a single int value.



2b) Image Processing

- **Memory Consumption using 2-D array**
- **References (Pointers) :**
  - H (4 Bytes for each)
- **Array length attribute:**
  - 1+H (4 Bytes each)
- **Total amount of unnecessary memory consumed**
  - 2H+1

  ~ 2H
- **Image data:**
  - WH (4 Bytes each)
- **Total (W=H=12)**
  - $4 * (2\mathrm{H} + 1 + \mathrm{WH}) = 676 \text{ bytes}$
  - Improvement by a factor 4.4

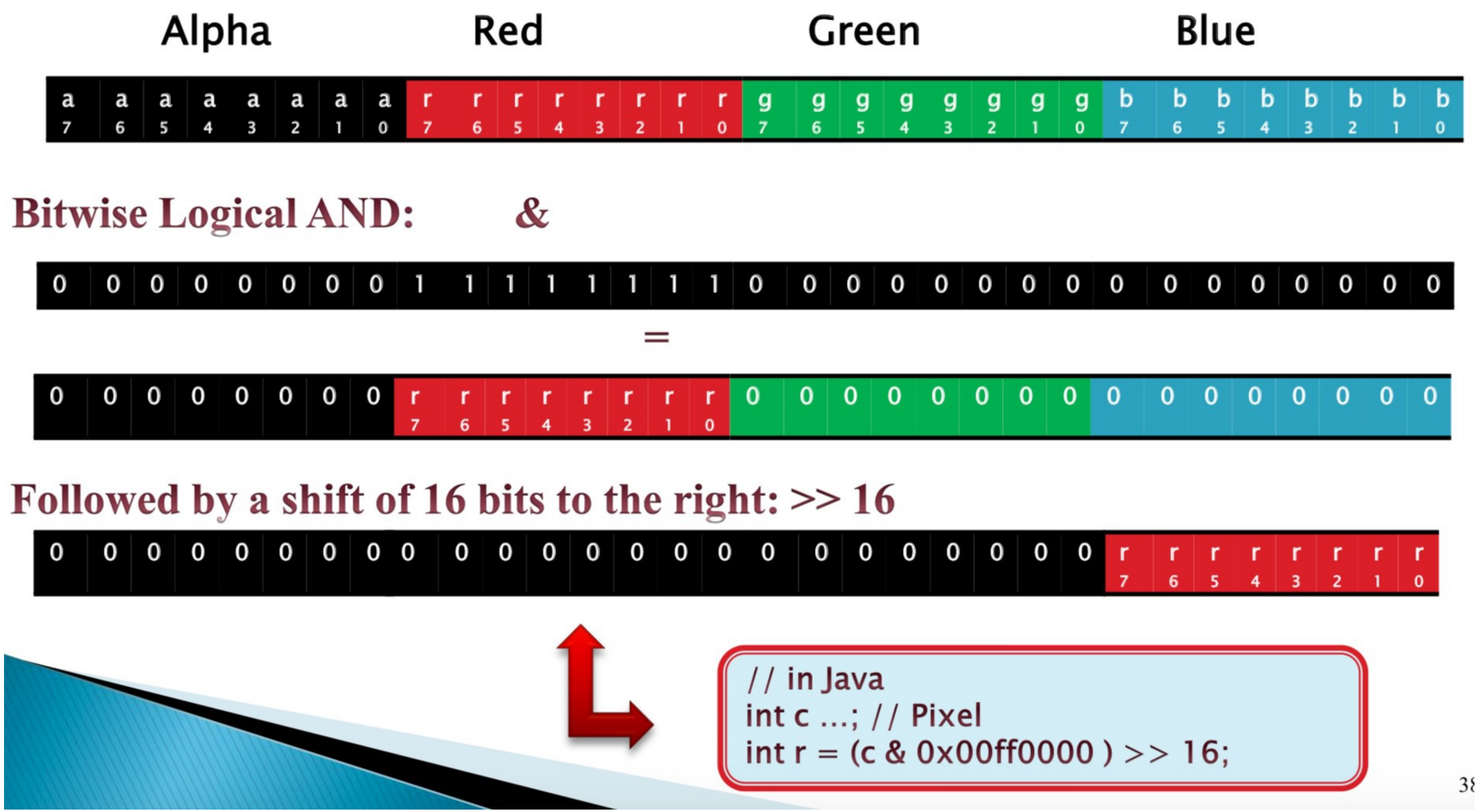
2c) Unpacking

Operations to pack and unpack a pixel

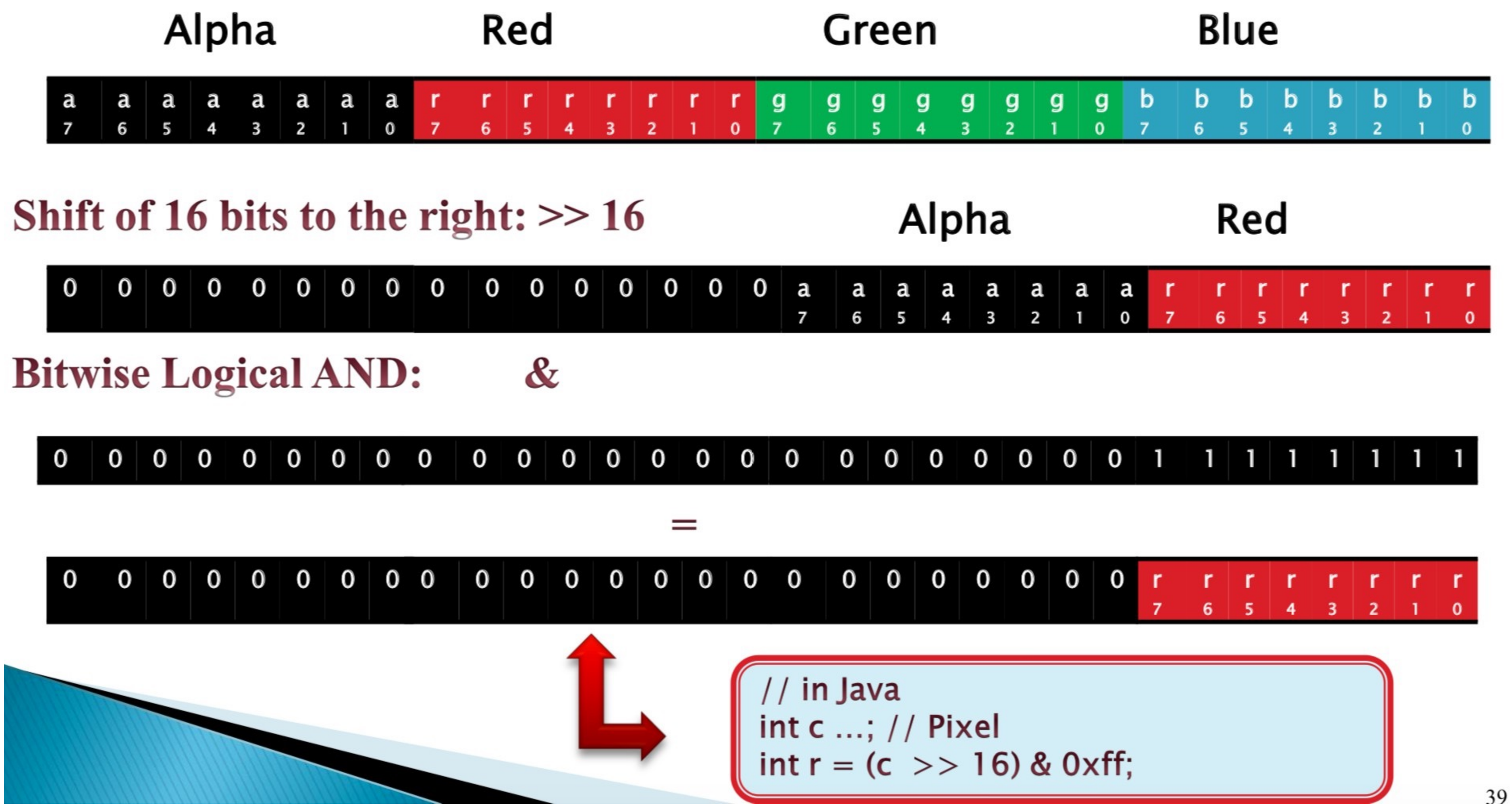
- **Right-shift operator** (**A>>B**); it shifts each bit of A to the right by B bits, and fills the least significant (rightmost) by zero.
- **Left-shift operator** (**A<<B**); it shifts each bit of A to the left by B bits, and fills the least significant (leftmost) by zero.

- **Bitwise logical operators:**
  - AND: &
  - OR : |
  - XOR: ^
  - Negation: ~

2c)

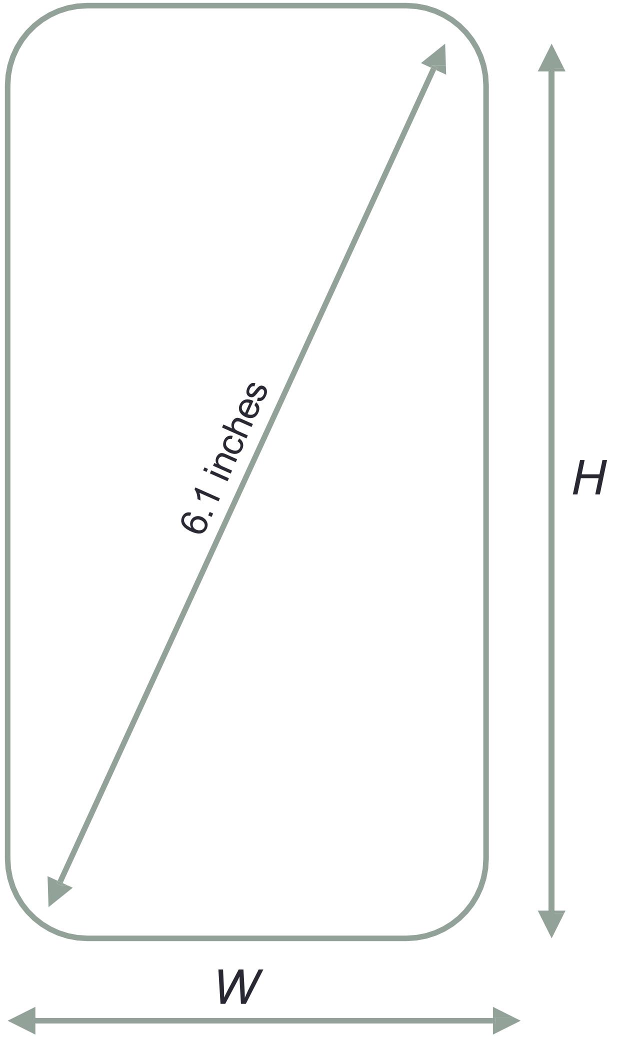

Alpha Red Green Blue
a7 a6 a5 a4 a3 a2 a1 a0 r7 r6 r5 r4 r3 r2 r1 r0 g7 g6 g5 g4 g3 g2 g1 g0 b7 b6 b5 b4 b3 b2 b1 b0
Bitwise Logical AND: &
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
=
0 0 0 0 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Followed by a shift of 16 bits to the right: >> 16
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0
// in Java
int c ...; // Pixel
int r = (c & 0x00ff0000) >> 16;
38

2c)


Alpha Red Green Blue
a7 a6 a5 a4 a3 a2 a1 a0 r7 r6 r5 r4 r3 r2 r1 r0 g7 g6 g5 g4 g3 g2 g1 g0 b7 b6 b5 b4 b3 b2 b1 b0
Shift of 16 bits to the right: >> 16
Alpha Red
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 a7 a6 a5 a4 a3 a2 a1 a0 r7 r6 r5 r4 r3 r2 r1 r0
Bitwise Logical AND: &
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
=
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 r7 r6 r5 r4 r3 r2 r1 r0
// in Java
int c ...; // Pixel
int r = (c >> 16) & 0xff;
39

2d) Image Resolution

- Screen Size (Diagonal) = 6.1 inches
- Aspect Ratio = 19.5 : 9 → H = (19.5/9) x W
- Resolution = 828 x 1792 pixels



According to Pythagora's Theorem:

$W^2 + \{ (19.5/9) \times W \}^2 = (6.1)^2$

Solving for W:

W = 2.5561 inches

Therefore,

2.5561 inches = 828 pixels

1 inch ≅ 324 pixels

3a) Uniform Quantization

- Quantization: For an Image I, containing m different colors $C = \{C_1, C_2, \ldots C_m\}$, where $m \le 2^{24}$ (For a 3 x 8 bit image), we have to replace the original colors with a set $C' = \{C_1, C_2, \ldots C_n\}$, with $n \ll m$ in such a way that we have minimal perceptible degradation in the resulting image.

- Uniform Quantization.
  - Each axis of the color space is divided into equal sized segments. (The number of these segments depends on storage requirements. For e.g. 8 for red, 8 for green and 4 for blue for 8 bit storage)
  - Each of the original colors is then mapped to the region which it falls in (256 for 8 bits).
  - Representative colors for each region is then the average of all the colors mapped to that region

3a) Median Cut Quanization

Source: **An Overview of Color Quantization Techniques by Steven Segenchuk**
https://web.cs.wpi.edu/~matt/courses/cs563/talks/color_quant/CQindex.html

- Median Cut Algorithm
  - Find the smallest box which contains all the colors in the image.
  - Sort the enclosed colors along the longest axis of the box.
  - Split the box into 2 regions at median of the sorted list.
  - Repeat the above process until the original color space has been divided into 256 regions.
  - The representative colors are found by averaging the colors in each box.

3b) (Random) Noise Dithering

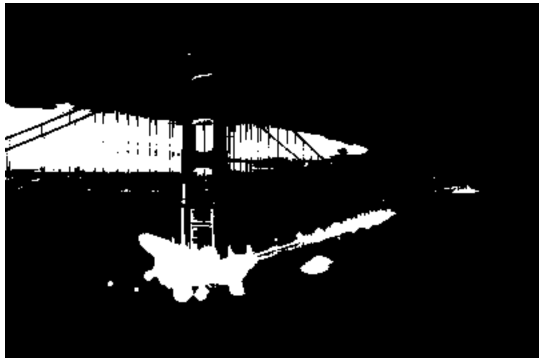
- Reduce effects of quantization(such as color banding) by adding uniformly distributed white noise (dither signal) to the input image prior to quantization.

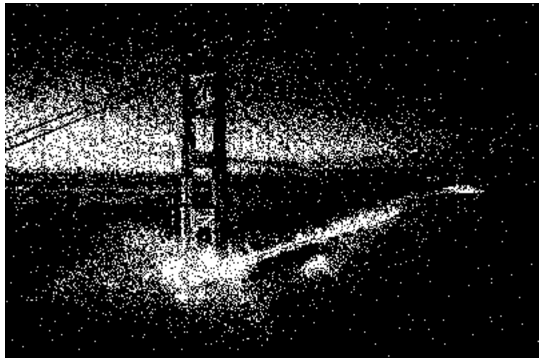
Quantization (Without Dithering):

$$P(x,y) = Q\big(I(x,y)\big) = floor(\frac{I(x,y)}{256}\,2^{b})$$

Quantization (With Dithering):

$$P(x,y) = Q(I(x,y) + noise(x,y))$$





source:https://surma.dev/things/ditherpunk/