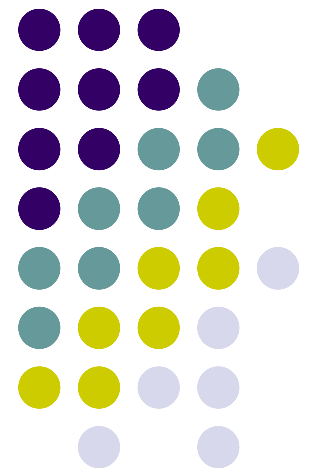
Digital Image Processing (CS/ECE 545) Lecture 9: Color Images (Part 2) & Introduction to Spectral Techniques (Fourier Transform, DFT, DCT)

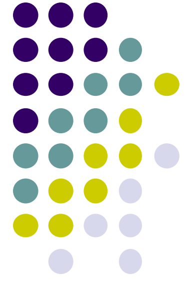
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

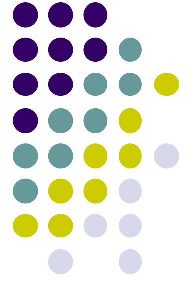


Organization of Color Images

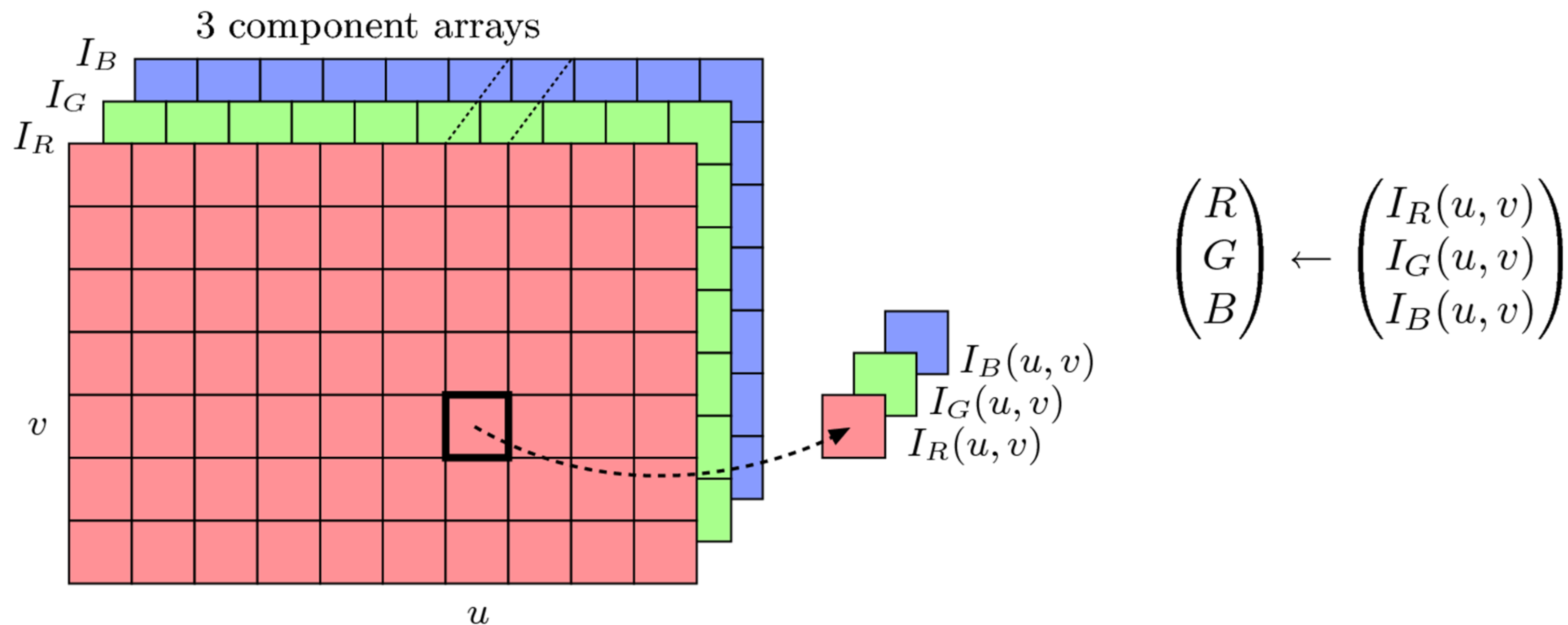


- **True color:** Uses all colors in color space
- **Indexed color:** Uses only some colors
  - Which subset of colors to use? Depends on application
- True color:
  - used in applications that contain many colors with subtle differences
  - E.g. digital photography or photorealistic rendering
- Two main ways to organize true color
  - Component ordering
  - Packed ordering

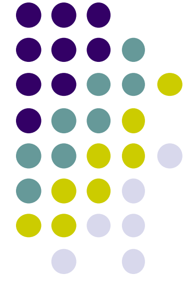
True Color: Component Ordering



- Colors in 3 separate arrays of similar length
- Retrieve same location (u,v) in each R, G and B array

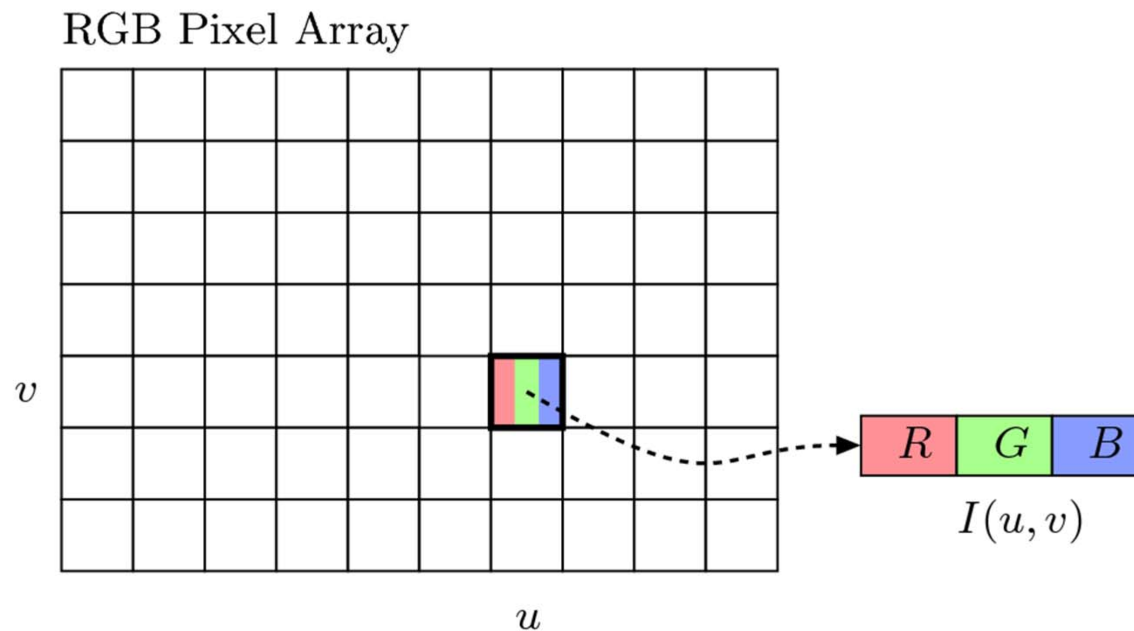


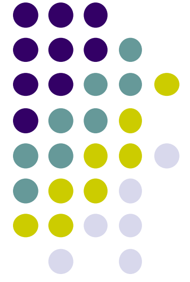
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} \leftarrow \begin{pmatrix} I_R(u,v) \\ I_G(u,v) \\ I_B(u,v) \end{pmatrix}$$



True Color: Packed Ordering

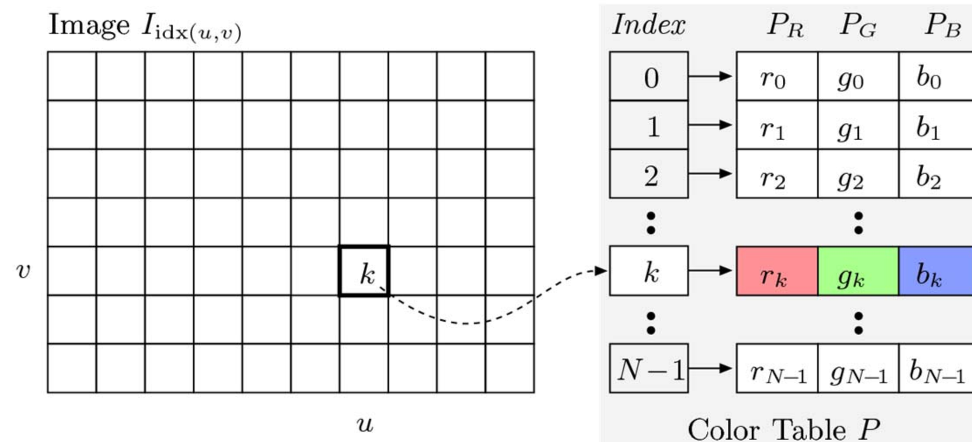
- Component (RGB) values representing a pixel's color is packed together into single element



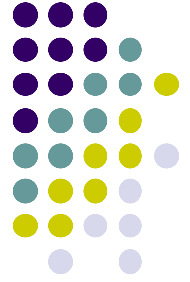


Indexed Images

- Permit only limited number of distinct colors (N = 2 to 256)
- Used in illustrations or graphics containing large regions with same color
- Instead of intensity values, image contains indices into color table or palette
- Palette saved as part of image
- Converting from true color to indexed color requires quantization

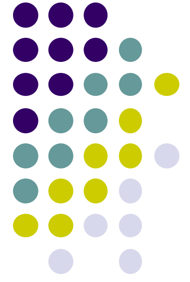


Color Images in ImageJ



- ImageJ supports 2 types of color images
  - RGB full-color images (24-bit “RGB color”),
    - packed order
    - Supports TIFF, BMP, JPEG, PNG and RAW file formats
  - Indexed images (“8-bit color”)
    - Up to 256 colors max (8 bits)
    - Supports GIF, PNG, BMP and TIFF (uncompressed) file formats
- See section 12.1.2 of Burger & Burge

Color Image Conversion in ImageJ



- Methods for converting between different types of color and grayscale image objects
- **Note:** if **doScaling** is **true**, pixel values scaled to maximum range of new image

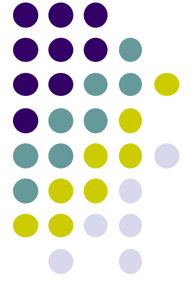
`ImageProcessor convertToByte(boolean doScaling)`
Converts to an 8-bit grayscale image (`ByteProcessor`).

`ImageProcessor convertToShort(boolean doScaling)`
Converts to a 16-bit grayscale image (`ShortProcessor`).

`ImageProcessor convertToFloat()`
Converts to a 32-bit floating-point image (`FloatProcessor`).

`ImageProcessor convertToRGB()`
Converts to a 32-bit RGB color image (`ColorProcessor`).

Conversion to ImagePlus Objects



- Do not create new image. Just modify original ImagePlus object

`ImageConverter(ImagePlus ipl)`
Instantiates an `ImageConverter` object for the image *ipl*.

`void convertToGray8()`
Converts *ipl* to an 8-bit grayscale image.

`void convertToGray16()`
Converts *ipl* to a 16-bit grayscale image.

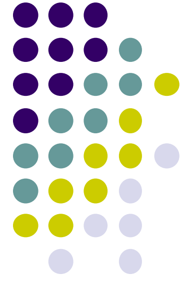
`void convertToGray32()`
Converts *ipl* to a 32-bit grayscale image (`float`).

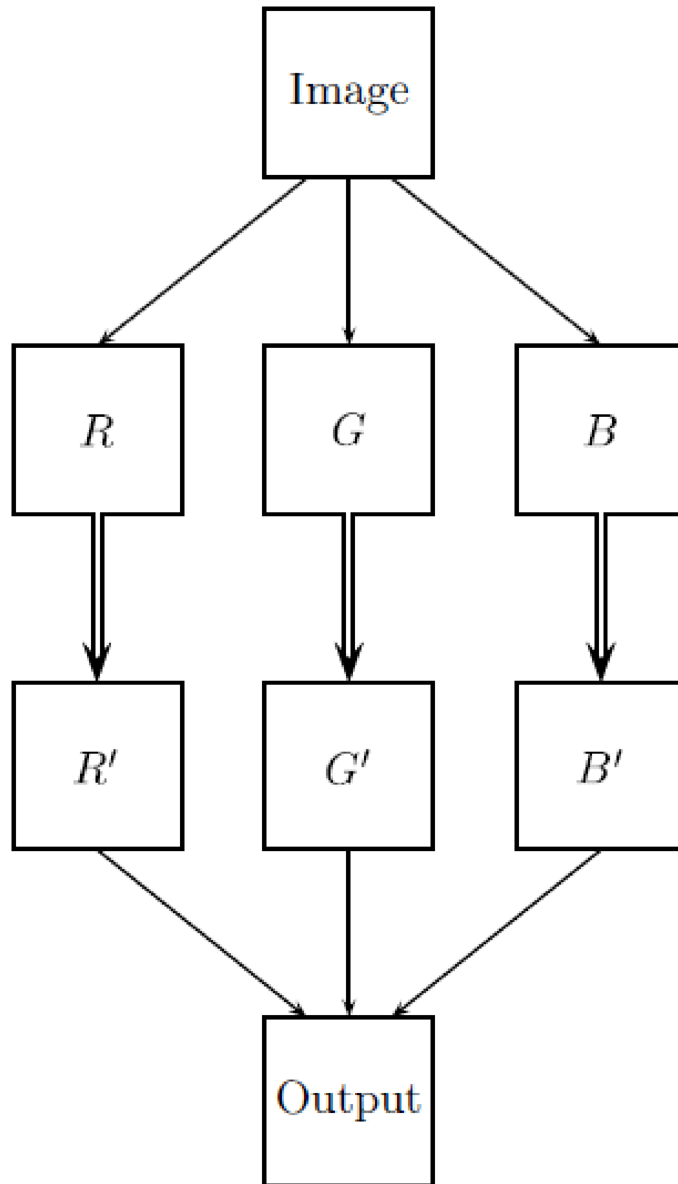
`void convertToRGB()`
Converts *ipl* to an RGB color image.

`void convertRGBtoIndexedColor(int nColors)`
Converts the RGB true color image *ipl* to an indexed image with 8-bit index values and *nColors* colors, performing color quantization.

`void convertToHSB()`
Converts *ipl* to a color image using the HSB color space (see Sec. 12.2.3).

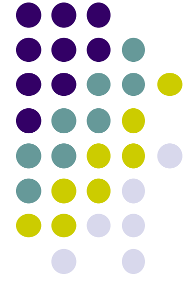
`void convertHSBToRGB()`
Converts an HSB color space image *ipl* to an RGB color image.

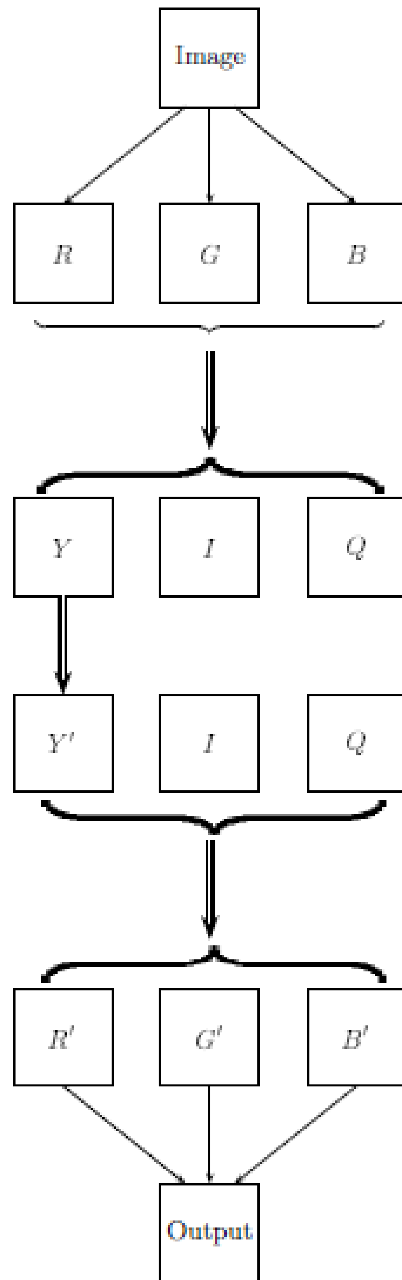




General Strategies for Processing Color Images

- **Strategy 1:** Process each RGB matrix separately

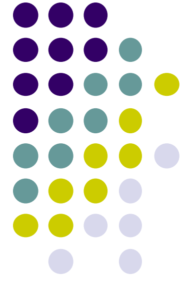




General Strategies for Processing Color Images

- **Strategy 2:** Compute luminance (weighted average of RGB), process intensity matrix

RGB to HSV Conversion



- Define the following values

$$C_{\text{high}} = \max(R, G, B), \quad C_{\text{low}} = \min(R, G, B), \quad \text{and}$$
$$C_{\text{rng}} = C_{\text{high}} - C_{\text{low}}.$$

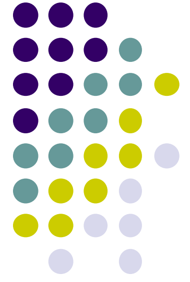
- Find **saturation** of RGB color components (C_{max}= 255)

$$S_{\text{HSV}} = \begin{cases} \frac{C_{\text{rng}}}{C_{\text{high}}} & \text{for } C_{\text{high}} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- And luminance value

$$V_{\text{HSV}} = \frac{C_{\text{high}}}{C_{\text{max}}}$$

RGB to HSV Conversion



- Normalize each component using

$$R' = \frac{C_{\text{high}} - R}{C_{\text{rng}}} \qquad G' = \frac{C_{\text{high}} - G}{C_{\text{rng}}} \qquad B' = \frac{C_{\text{high}} - B}{C_{\text{rng}}}$$

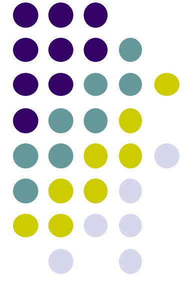
- Calculate preliminary hue value *H'* as

$$H' = \begin{cases} B' - G' & \text{if } R = C_{\text{high}} \\ R' - B' + 2 & \text{if } G = C_{\text{high}} \\ G' - R' + 4 & \text{if } B = C_{\text{high}} \end{cases}$$

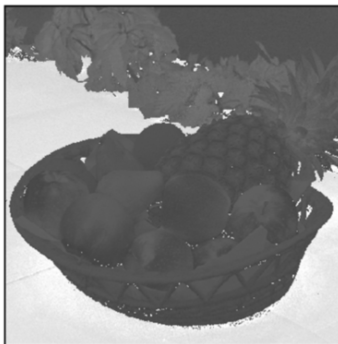
- Finally, obtain final hue value by normalizing to interval [0,1]

$$H_{\text{HSV}} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & \text{for } H' < 0 \\ H' & \text{otherwise} \end{cases}$$

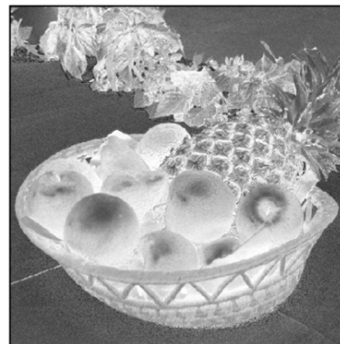
Example: RGB to HSV Conversion



Original RGB image



H_{HSV}

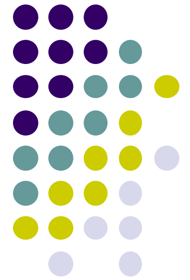


S_{HSV}

V_{HSV}

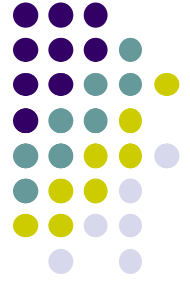
HSV values in grayscale

RGB to HSV Conversion Java Code



```
1   static float[] RGBtoHSV (int R, int G, int B, float[] HSV) {
2     // R, G, B ∈ [0, 255]
3     float H = 0, S = 0, V = 0;
4     float cMax = 255.0f;
5     int cHi = Math.max(R,Math.max(G,B)); // highest color value
6     int cLo = Math.min(R,Math.min(G,B)); // lowest color value
7     int cRng = cHi - cLo;          // color range
8
9     // compute value V
10    V = cHi / cMax;
11
12    // compute saturation S
13    if (cHi > 0)
14      S = (float) cRng / cHi;
15
16    // compute hue H
17    if (cRng > 0) { // hue is defined only for color pixels
18      float rr = (float)(cHi - R) / cRng;
19      float gg = (float)(cHi - G) / cRng;
20      float bb = (float)(cHi - B) / cRng;
21      float hh;
22      if (R == cHi)                      // R is highest color value
23        hh = bb - gg;
24      else if (G == cHi)                 // G is highest color value
25        hh = rr - bb + 2.0f;
26      else                               // B is highest color value
27        hh = gg - rr + 4.0f;
28      if (hh < 0)
29        hh= hh + 6;
30      H = hh / 6;
31    }
32
33    if (HSV == null) // create a new HSV array if needed
34      HSV = new float[3];
35    HSV[0] = H; HSV[1] = S; HSV[2] = V;
36    return HSV;
37  }
```

HSV to RGB Conversion



- To convert HSV tuple with each component in [0,1] range into RGB, first calculate

$$H' = (6 \cdot H_{\mathrm{HSV}}) \bmod 6$$

- Then compute the following intermediate values

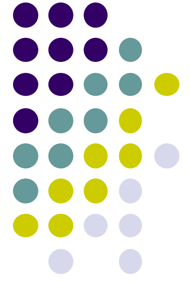
$$\begin{aligned} c_1 &= \lfloor H' \rfloor, & x &= (1 - S_{\mathrm{HSV}}) \cdot V_{\mathrm{HSV}}, \\ c_2 &= H' - c_1, & y &= (1 - (S_{\mathrm{HSV}} \cdot c_2)) \cdot V_{\mathrm{HSV}}, \\ & & z &= (1 - (S_{\mathrm{HSV}} \cdot (1 - c_2))) \cdot V_{\mathrm{HSV}}. \end{aligned}$$

where $v = V_{HSV}$

- Normalized RGB values (in range [0,1]) are calculated as

$$(R', G', B') = \begin{cases} (v, z, x) & \text{if } c_1 = 0 \\ (y, v, x) & \text{if } c_1 = 1 \\ (x, v, z) & \text{if } c_1 = 2 \\ (x, y, v) & \text{if } c_1 = 3 \\ (z, x, v) & \text{if } c_1 = 4 \\ (v, x, y) & \text{if } c_1 = 5. \end{cases}$$

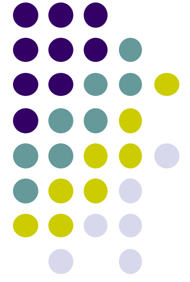
HSV to RGB Conversion



- RGB Components can be scaled to whole numbers in range [0,255] as

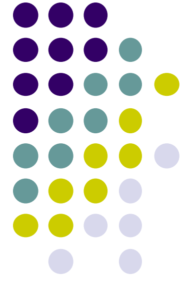
$$
\begin{aligned}
R &= \min\big(\mathrm{round}(255 \cdot R'), 255\big), \\
G &= \min\big(\mathrm{round}(255 \cdot G'), 255\big), \\
B &= \min\big(\mathrm{round}(255 \cdot B'), 255\big).
\end{aligned}
$$

HSV to RGB Code



```
1   public static int HSVtoRGB (float h, float s, float v) {
2     // h, s, v ∈ [0,1]
3     float r = 0, g = 0, b = 0;
4     float hh = (6 * h) % 6;            // h' ← (6 · h) mod 6
5     int   c1 = (int) hh;               // c1 ← ⌊h'⌋
6     float c2 = hh - c1;
7     float x = (1 - s) * v;
8     float y = (1 - (s * c2)) * v;
9     float z = (1 - (s * (1 - c2))) * v;
10    switch (c1) {
11      case 0: r = v; g = z; b = x; break;
12      case 1: r = y; g = v; b = x; break;
13      case 2: r = x; g = v; b = z; break;
14      case 3: r = x; g = y; b = v; break;
15      case 4: r = z; g = x; b = v; break;
16      case 5: r = v; g = x; b = y; break;
17    }
18    int R = Math.min((int)(r * 255), 255);
19    int G = Math.min((int)(g * 255), 255); // Eqn. (12.19)
20    int B = Math.min((int)(b * 255), 255);
21    // create int-packed RGB-color:
22    int rgb = ((R & 0xff)<<16) | ((G & 0xff)< 8) | B & 0xff;
23    return rgb;
24  }
```

RGB to HLS Conversion



- Compute Hue same way as for HSV model

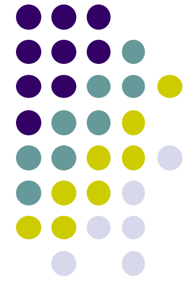
$$H_{\text{HLS}} = H_{\text{HSV}}$$

- Then compute the other 2 values as:

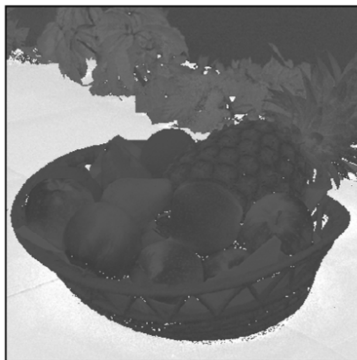
$$L_{\text{HLS}} = \frac{(C_{\text{high}} + C_{\text{low}})/255}{2}$$

$$S_{\text{HLS}} = \begin{cases} 0 & \text{for } \; L_{\text{HLS}} = 0 \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{L_{\text{HLS}}} & \text{for } \; 0 < L_{\text{HLS}} \leq 0.5 \\ 0.5 \cdot \frac{C_{\text{rng}}/255}{1-L_{\text{HLS}}} & \text{for } \; 0.5 < L_{\text{HLS}} < 1 \\ 0 & \text{for } \; L_{\text{HLS}} = 1. \end{cases}$$

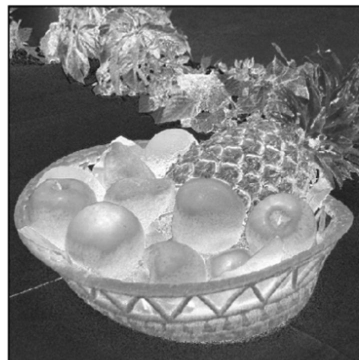
Example RGB to HLS Conversion



Original RGB image



H_{HLS}

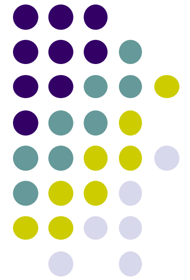


S_{HLS}

L_{HLS}

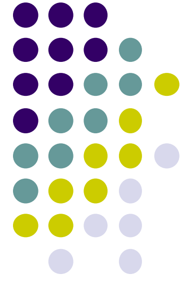
HLS values in grayscale

RGB to HLS Conversion Code



```
1  static float[] RGBtoHLS (float R, float G, float B) {
2    // R, G, B assumed to be in [0, 1]
3    float cHi = Math.max(R,Math.max(G,B)); // highest color value
4    float cLo = Math.min(R,Math.min(G,B)); // lowest color value
5    float cRng = cHi - cLo;          // color range
6
7    // compute luminance L
8    float L = (cHi + cLo)/2;
9
10   // compute saturation S
11   float S = 0;
12   if (0 < L && L < 1) {
13     float d = (L <= 0.5f) ? L : (1 - L);
14     S = 0.5f * cRng / d;
15   }
16
17   // compute hue H
18   float H=0;
19   if (cHi > 0 && cRng > 0) {       // a color pixel
20     float rr = (float)(cHi - R) / cRng;
21     float gg = (float)(cHi - G) / cRng;
22     float bb = (float)(cHi - B) / cRng;
23     float hh;
24     if (R == cHi)                  // R is highest color value
25       hh = bb - gg;
26     else if (G == cHi)             // G is highest color value
27       hh = rr - bb + 2.0f;
28     else                           // B is highest color value
29       hh = gg - rr + 4.0f;
30
31     if (hh < 0)
32       hh= hh + 6;
33     H = hh / 6;
34   }
35
36   return new float[] {H,L,S};
37 }
```

HLS to RGB Conversion



- Assuming H, L and S in [0,1] range

$$(R', G', B') = \begin{cases} (0,0,0) & \text{for } L_{\mathrm{HLS}} = 0 \\ (1,1,1) & \text{for } L_{\mathrm{HLS}} = 1 \end{cases}$$

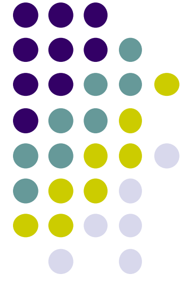
- Otherwise, calculate

$$H' = (6 \cdot H_{\mathrm{HLS}}) \bmod 6$$

- Then calculate the values

$$\begin{aligned} c_1 &= \lfloor H' \rfloor & \qquad d &= \begin{cases} S_{\mathrm{HLS}} \cdot L_{\mathrm{HLS}} & \text{for } L_{\mathrm{HLS}} \leq 0.5 \\ S_{\mathrm{HLS}} \cdot (1 - L_{\mathrm{HLS}}) & \text{for } L_{\mathrm{HLS}} > 0.5 \end{cases} \\ c_2 &= H' - c_1 & & \\ w &= L_{\mathrm{HLS}} + d & \qquad y &= w - (w - x) \cdot c_2 \\ x &= L_{\mathrm{HLS}} - d & \qquad z &= x + (w - x) \cdot c_2 . \end{aligned} \tag{12.25}$$

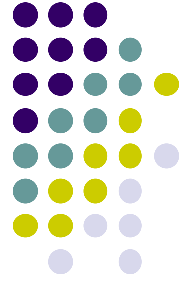
HLS to RGB Conversion



- Assignment of RGB values is done as follows

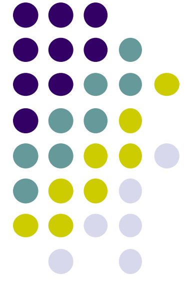
$$(R', G', B') = \begin{cases} (w, z, x) & \text{if } \; c_1 = 0 \\ (y, w, x) & \text{if } \; c_1 = 1 \\ (x, w, z) & \text{if } \; c_1 = 2 \\ (x, y, w) & \text{if } \; c_1 = 3 \\ (z, x, w) & \text{if } \; c_1 = 4 \\ (w, x, y) & \text{if } \; c_1 = 5 \end{cases}$$

HLS to RGB Code



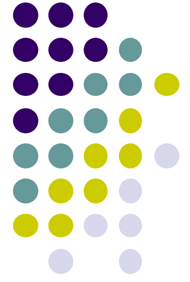
```
1  static float[] HLStoRGB (float H, float L, float S) {
2    // H, L, S assumed to be in [0,1]
3    float R = 0, G = 0, B = 0;
4
5    if (L <= 0)        // black
6      R = G = B = 0;
7    else if (L >= 1)   // white
8      R = G = B = 1;
9    else {
10     float hh = (6 * H) % 6;
11     int   c1 = (int) hh;
12     float c2 = hh - c1;
13     float d = (L <= 0.5f) ? (S * L) : (S * (1 - L));
14     float w = L + d;
15     float x = L - d;
16     float y = w - (w - x) * c2;
17     float z = x + (w - x) * c2;
18     switch (c1) {
19       case 0: R=w; G=z; B=x; break;
20       case 1: R=y; G=w; B=x; break;
21       case 2: R=x; G=w; B=z; break;
22       case 3: R=x; G=y; B=w; break;
23       case 4: R=z; G=x; B=w; break;
24       case 5: R=w; G=x; B=y; break;
25     }
26   }
27   return new float[] {R,G,B};
28 }
```

TV Color Spaces – YUV, YIQ, YC_bC_r



- YUV, YIQ: color encoding for analog NTSC and PAL
- YC_bC_r: Digital TV encoding
- Key common ideas:
  - Separate luminance component Y from 2 chroma components
  - Instead of encoding colors, encode color differences between components (maintains compatibility with black and white TV)

YUV



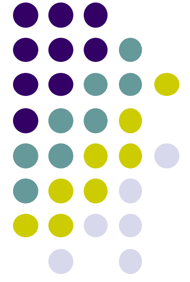
- Basis for color encoding in analog TV in north america (NTSC) and Europe (PAL)
- Y components computed from RGB components as

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

- UV components computed as:

$$U = 0.492 \cdot (B - Y) \qquad \text{and} \qquad V = 0.877 \cdot (R - Y)$$

YUV



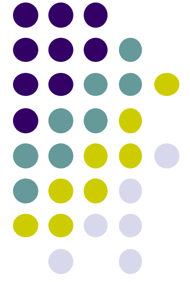
- Entire transformation from RGB to YUV

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

- Invert matrix above to transform from YUV back to RGB

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

YIQ



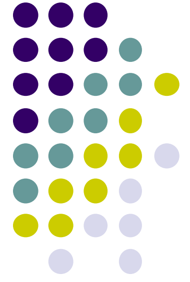
- Original NTSC used variant of YUV called YIQ
- *Y* component is same as in YUV
- Both U and V color vectors rotated and mirrored so that

$$\begin{pmatrix} I \\ Q \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} U \\ V \end{pmatrix}$$

2D rotation matrix

where β = 0.576 (33 degrees)

YC_bC_r



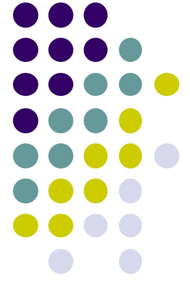
- Internationally standardized variant of YUV
- Used for digital TV and image compression (e.g. JPEG)
- Y, C_b, C_r components calculated as

$$Y = w_R \cdot R + (1 - w_B - w_R) \cdot G + w_B \cdot B$$
$$C_b = \frac{0.5}{1 - w_B} \cdot (B - Y)$$
$$C_r = \frac{0.5}{1 - w_R} \cdot (R - Y)$$

- Inverse transform from YC_bC_r to RGB

$$R = Y + \frac{1 - w_R}{0.5} \cdot C_r$$
$$G = Y - \frac{w_B \cdot (1 - w_B) \cdot C_b \; - \; w_R \cdot (1 - w_R) \cdot C_r}{0.5 \cdot (1 - w_B - w_R)}$$
$$B = Y + \frac{1 - w_B}{0.5} \cdot C_b$$

YC_bC_r



- ITU recommendation BT.601 specifies values:

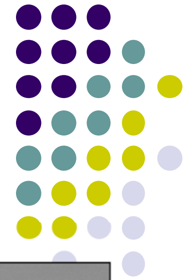
  $w_R = 0.299, \quad w_B = 0.114, \quad w_G = 1 - w_B - w_R = 0.587$

- Thus the transformation

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

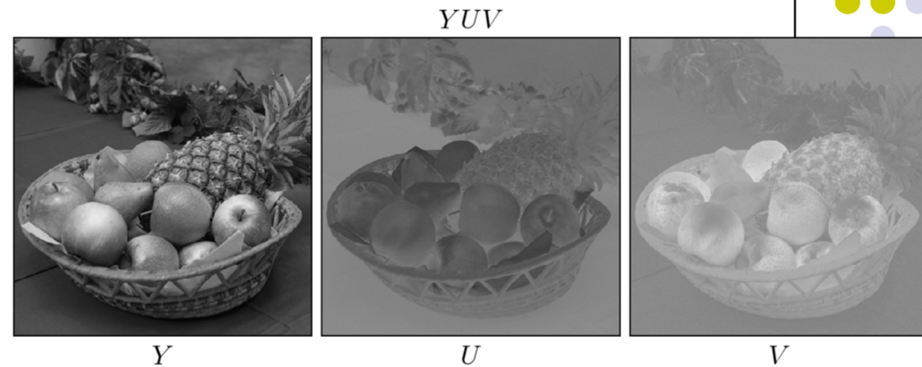
- And the inverse transformation becomes

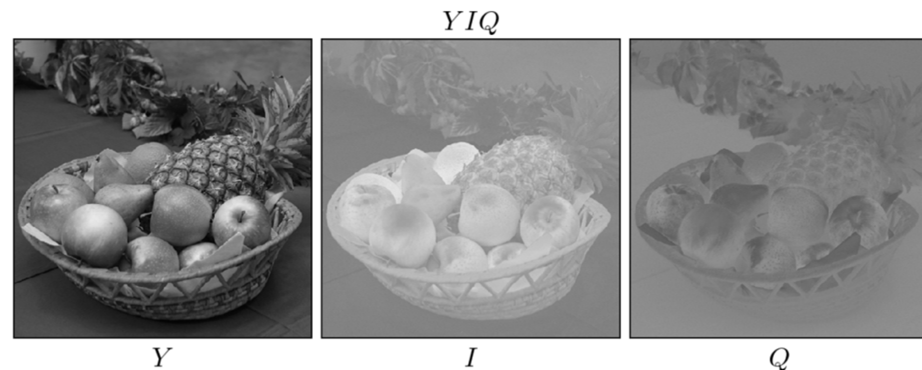
$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.000 & 0.000 & 1.403 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.773 & 0.000 \end{pmatrix} \cdot \begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix}$$

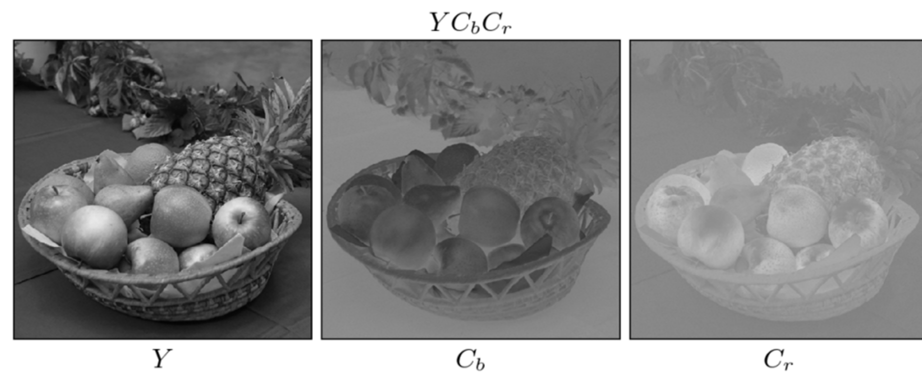


Comparing YUV, YIQ and YC_bC_r

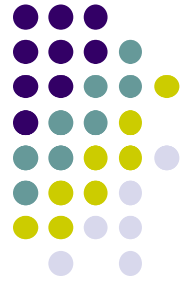
- Y values are identical in all 3 color spaces



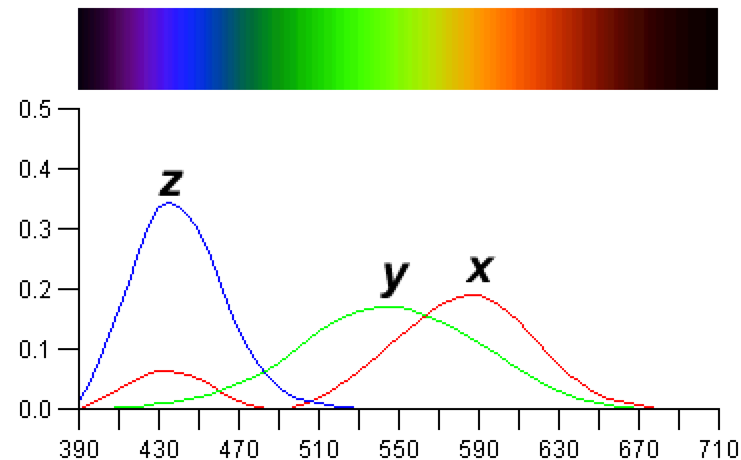




CIE Color Space



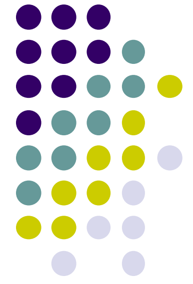
- *CIE* (Commission Internationale d'Eclairage) came up with 3 hypothetical lights X, Y, and Z with these spectra:



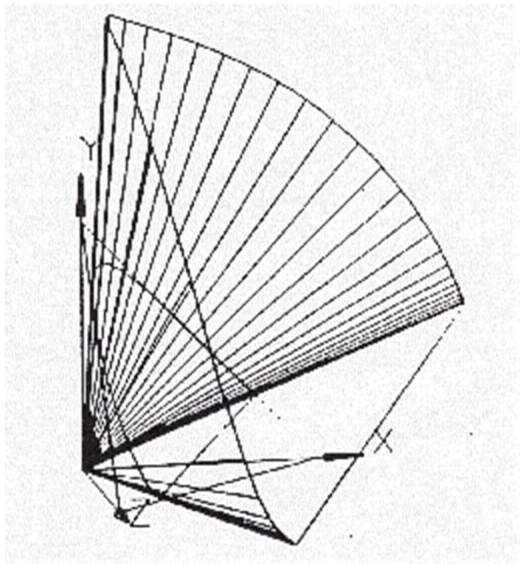
Note that:
X ~ R
Y ~ G
Z ~ B

- **Idea:** any wavelength λ can be matched perceptually by *positive* combinations of X,Y,Z
- CIE created table of XYZ values for all *visible* colors

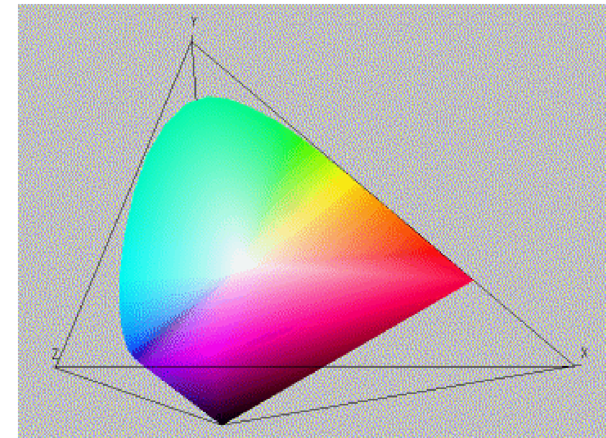
CIE Color Space



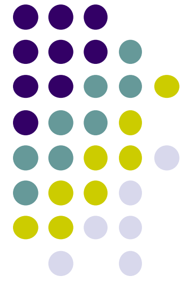
- The *gamut* of all colors perceivable is thus a three-dimensional shape in X,Y,Z
- Color = X'*X* + Y'*Y* + Z'*Z*

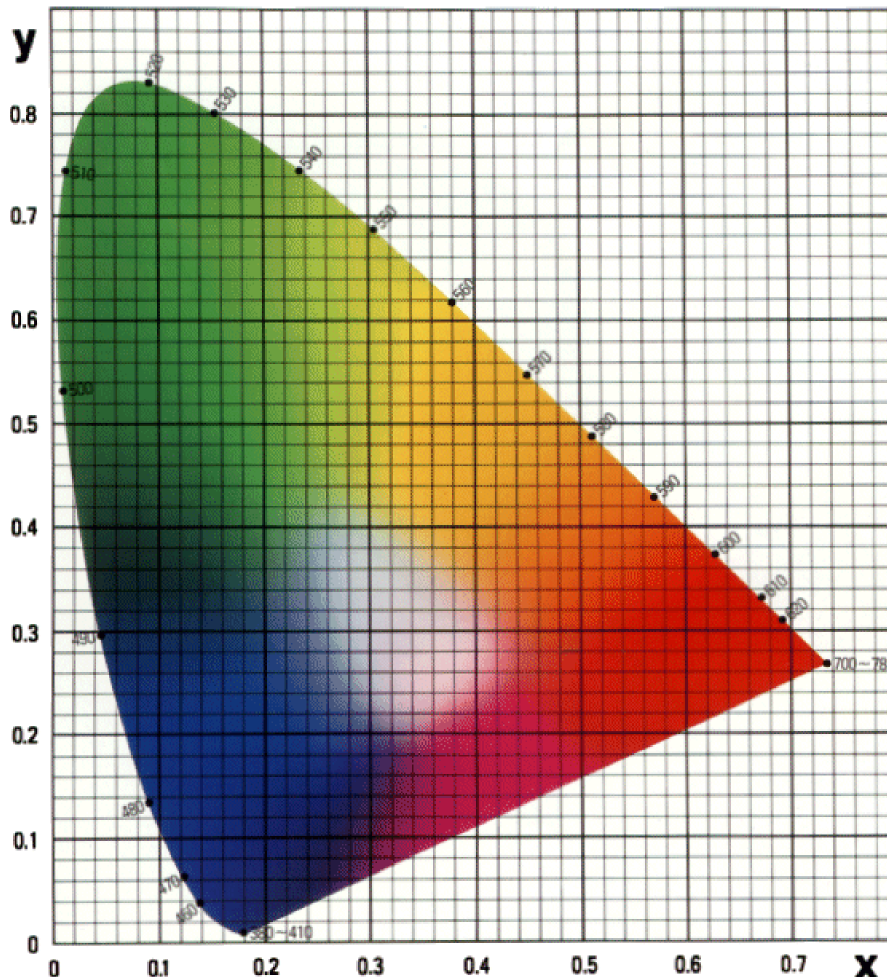


Human Perceptual Gamut



CIE Chromaticity Diagram (1931)





- For simplicity, we often project to the 2D plane
- Also normalize

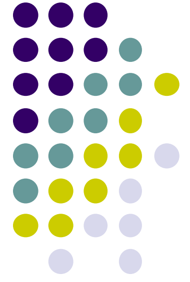
$$x + y + z = 1$$

$$x = \frac{X}{X+Y+Z} \qquad y = \frac{Y}{X+Y+Z} \qquad z = \frac{Z}{X+Y+Z}$$

- Note: Inside horseshoe visible, outside invisible to eye

Note: Look up *x, y*
Calculate z as *1 – x - y*

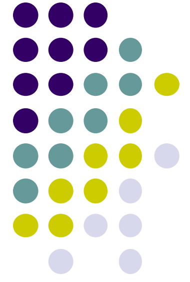
Standard Illuminants



- Central goal of CIE chart is the quantitative measurement of colors in physical reality
- CIE specifies some standard illuminants for many **real** and **hypothetical** light sources
- Specified by spectral radiant power distribution and correlated color temperature
- **D50:** natural direct sunlight
- **D65:** Indirect daylight, overcast sky

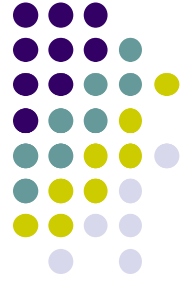
Pt.	Temp.	X	Y	Z	x	y
D50	5000° K	0.964296	1.000000	0.825105	0.3457	0.3585
D65	6500° K	0.950456	1.000000	1.088754	0.3127	0.3290
E	5400° K	1	1	1	1/3	1/3

CIE uses



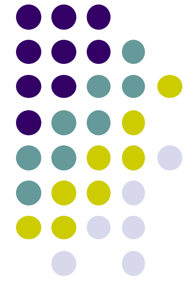
- Find complementary colors:
  - equal linear distances from white in opposite directions
- Measure hue and saturation:
  - Extend line from color to white till it cuts horseshoe (hue)
  - Saturation is ratio of distances color-to-white/hue-to-white
- Define and compare device color gamut (color ranges)
- **Problem:** not perceptually uniform:
  - Same amount of changes in different directions generate perceived difference that are not equal
  - CIE LUV, L*a*b* - uniform

CIE L*a*b*

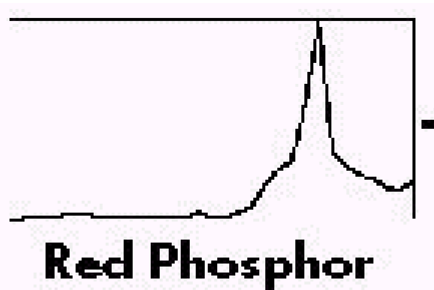


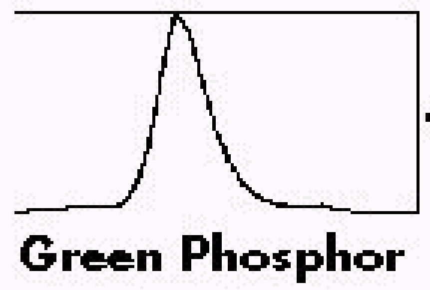
- Main goal was to make changes in this space linear with respect to human perception
- Now popular in high-quality photographic applications
- Used in Adobe photoshop as standard for converting between different color spaces
- Components:
  - Luminosity **L**
  - Color components **a*** and **b*** which specify color hue and saturation along green-red and blue-yellow axes
- All 3 components are measured relative to reference white
- Non-linear correction function (like gamma correction) applied

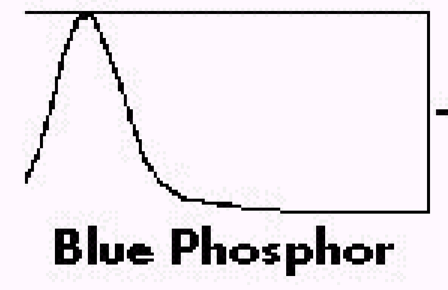
Device Color Gamuts



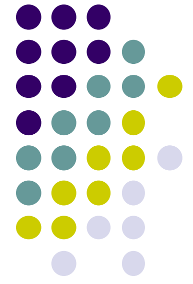
- Since X, Y, and Z are hypothetical light sources, no real device can produce the entire gamut of perceivable color
- Depends on physical means of producing color on device
- Example: R,G,B phosphors on CRT monitor



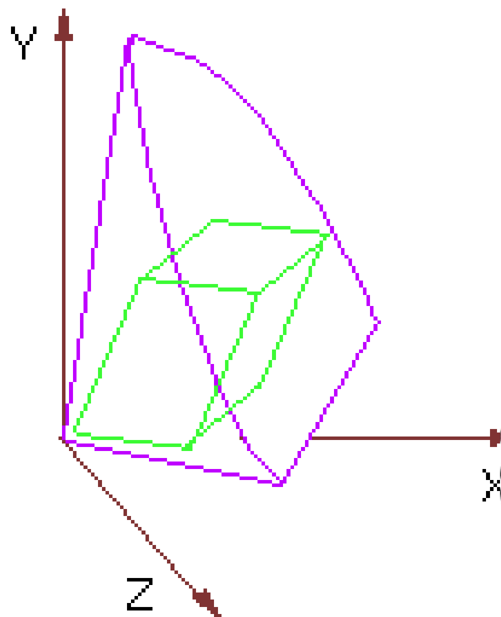


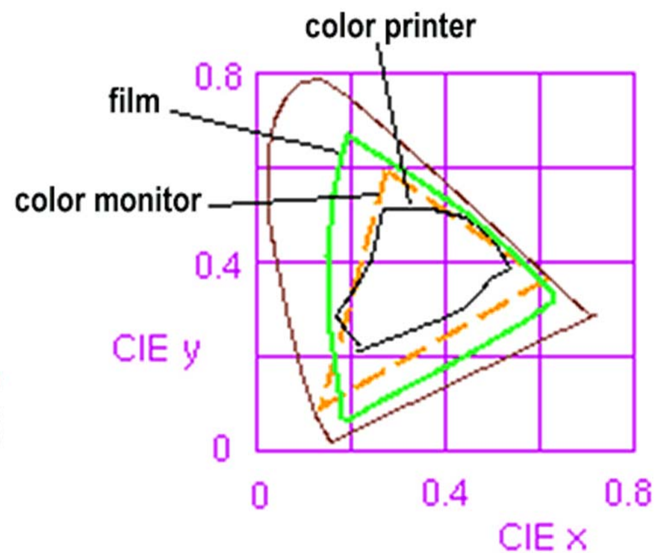


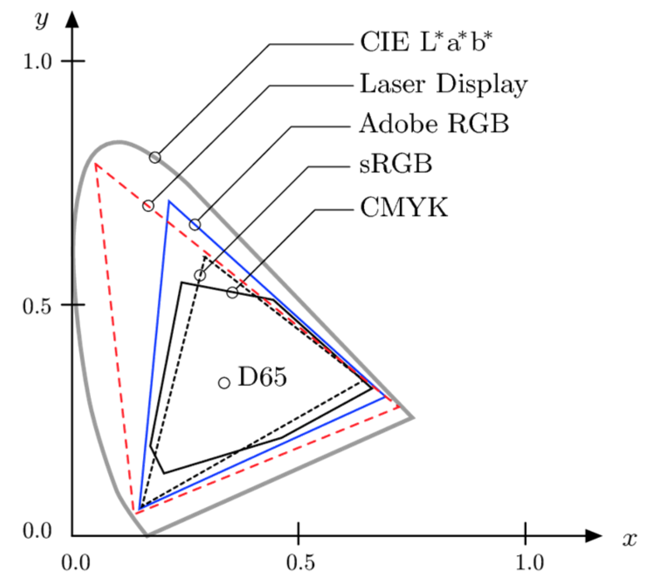
Device Color Gamuts



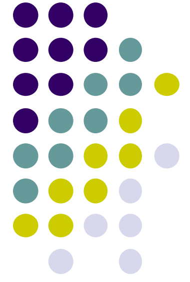
- The RGB color cube sits within CIE color space
- We can use the CIE chromaticity diagram to compare the gamuts of various devices
- E.g. compare color printer and monitor color gamuts







Transformation CIE XYZ to L*a*b*



- Current ISO Standard 13655 conversion

$$L^* = 116 \cdot Y' - 16$$
$$a^* = 500 \cdot (X' - Y')$$
$$b^* = 200 \cdot (Y' - Z')$$

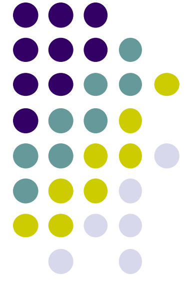
where

$$X' = f_1\left(\frac{X}{X_{\text{ref}}}\right) \quad Y' = f_1\left(\frac{Y}{Y_{\text{ref}}}\right) \quad Z' = f_1\left(\frac{Z}{Z_{\text{ref}}}\right)$$

$$f_1(c) = \begin{cases} c^{\frac{1}{3}} & \text{for } c > 0.008856 \\ 7.787 \cdot c + \frac{16}{116} & \text{for } c \leq 0.008856 \end{cases}$$

- Usually, standard illuminant D65 is specified as reference (X_{ref}, Y_{ref}, Z_{ref})
- Possible values of *a** and *b** are in range [-127, +127]

Transformation L*a*b* to CIE XYZ



- Reverse transformation from L*a*b* space to XYZ is

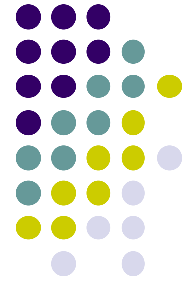
$$
\begin{aligned}
X &= X_{\text{ref}} \cdot f_2\left(\tfrac{a^*}{500} + Y'\right) \\
Y &= Y_{\text{ref}} \cdot f_2\left(Y'\right) \\
Z &= Z_{\text{ref}} \cdot f_2\left(Y' - \tfrac{b^*}{200}\right)
\end{aligned}
$$

where

$$
Y' = \tfrac{L^*+16}{116}
$$

$$
f_2(c) = \begin{cases} c^3 & \text{for } c^3 > 0.008856 \\ \frac{c-16/116}{7.787} & \text{for } c^3 \leq 0.008856 \end{cases}
$$

Example of L*a*b* Components

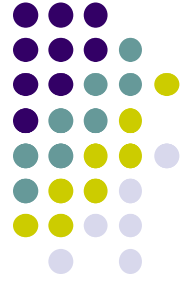


L^*

a^*

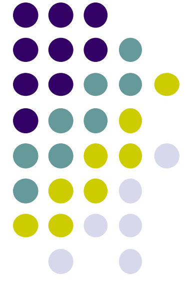
b^*

Code for XYZ to L*a*b* and L*a*b* to XYZ Conversion



```
23  // XYZ→CIELab: returns D65-related L*a*b values
24  // from D50-related XYZ values:
25  public float[] fromCIEXYZ(float[] XYZ50) {
26    float[] XYZ65 = catD50toD65.apply(XYZ50);
27    double xx = f1(XYZ65[0] / Xref);
28    double yy = f1(XYZ65[1] / Yref);
29    double zz = f1(XYZ65[2] / Zref);
30
31    float L = (float)(116 * yy - 16);
32    float a = (float)(500 * (xx - yy));
33    float b = (float)(200 * (yy - zz));
34    return new float[] {L, a, b};
35  }
36
37  // CIELab→XYZ: returns D50-related XYZ values
38  // from D65-related L*a*b* values:
39  public float[] toCIEXYZ(float[] Lab) {
40    double yy = ( Lab[0] + 16 ) / 116;
41    float X65 = (float) (Xref * f2(Lab[1] / 500 + yy));
42    float Y65 = (float) (Yref * f2(yy));
43    float Z65 = (float) (Zref * f2(yy - Lab[2] / 200));
44    float[] XYZ65 = new float[] {X65, Y65, Z65};
45    return catD65toD50.apply(XYZ65);
46  }
```

Measuring Color Differences

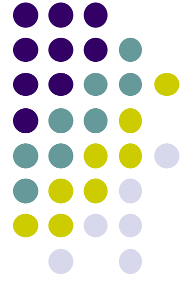


- Due to its uniformity with respect to human perception, differences between colors in L*a*b* color space can be determined as **euclidean distance**

$$\mathrm{ColorDist}_{\mathrm{Lab}}(\mathbf{C}_1, \mathbf{C}_2) = \|\mathbf{C}_1 - \mathbf{C}_2\|$$

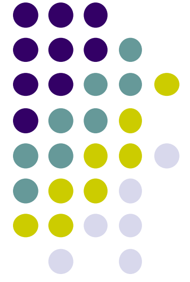
$$= \sqrt{(L_1^* - L_2^*)^2 + (a_1^* - a_2^*)^2 + (b_1^* - b_2^*)^2}$$

sRGB



- For many computer display-oriented applications, use of CIE color space may be too cumbersome
- sRGB
  - developed by Hewlett Packard and Microsoft
  - has relatively small gamut compared to L*a*b*
  - Its colors can be reproduced by most computer monitors
  - De Facto standard for digital cameras
- Several image formats (EXIF, PNG) based on sRGB

Transformation CIE XYZ to sRGB



- First compute linear RGB values as

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \boldsymbol{M}_{\mathrm{RGB}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

Where

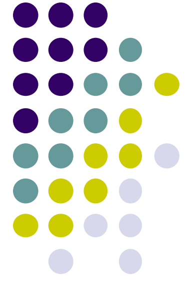
$$\boldsymbol{M}_{\mathrm{RGB}} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix}$$

- Next gamma correct linear RGB values

$$R' = f_\gamma(R) \qquad G' = f_\gamma(G) \qquad B' = f_\gamma(B)$$

$$f_\gamma(c) = \begin{cases} 1.055 \cdot c^{\frac{1}{2.4}} - 0.055 & \text{for } c > 0.0031308 \\ 12.92 \cdot c & \text{for } c \leq 0.0031308 \end{cases}$$

Transformation sRGB to CIE XYZ



- First linearize R' G' B' values as

$$R = f_\gamma^{-1}(R') \qquad G = f_\gamma^{-1}(G') \qquad B = f_\gamma^{-1}(B')$$

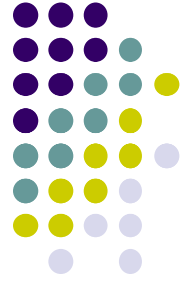
With

$$f_\gamma^{-1}(c') = \begin{cases} \left(\frac{c'+0.055}{1.055}\right)^{2.4} & \text{for } c' > 0.03928 \\ \frac{c'}{12.92} & \text{for } c' \le 0.03928 \end{cases}$$

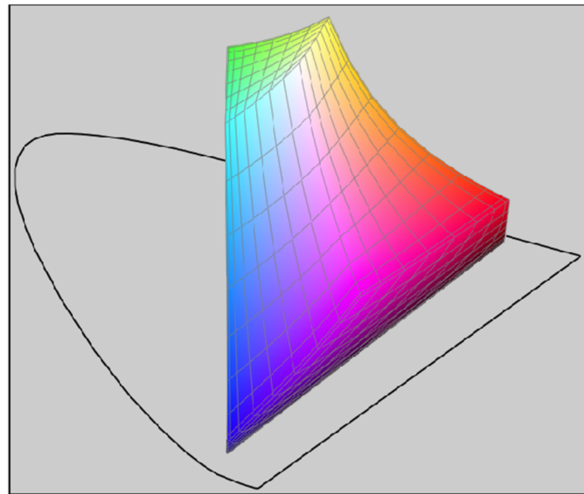
- Linearized RGB then transformed to XYZ as

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M_{\mathrm{RGB}}^{-1} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \qquad M_{\mathrm{RGB}}^{-1} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix}$$

Adobe RGB

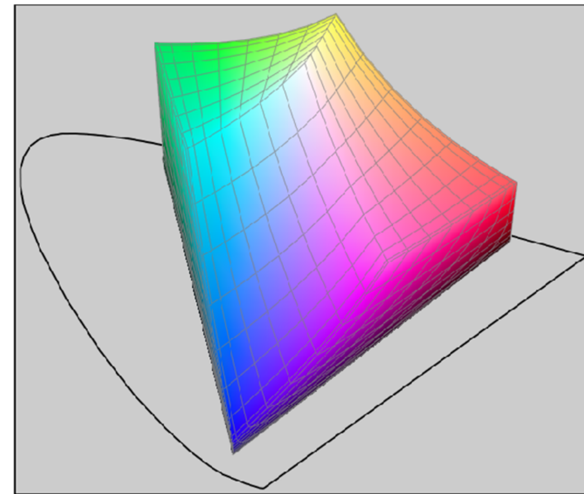


- Small gamut limited to colors reproducible by computer monitors is a weakness of sRGB
- Creates problems in areas such as printing
- Adobe RGB similar to sRGB but with much larger gamut



(a)

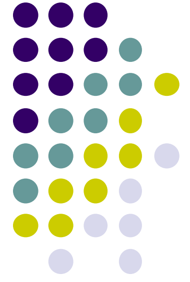
sRGB gamut



(b)

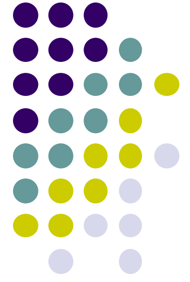
Adobe RGB gamut

Chromatic Adaptation



- Human eye adapts to make color of object same under different lighting conditions
- E.g. Paper appears white in bright daylight and under flourescent light
- CIE color system allows colors to be specified relative to white point, called **relative colorimetry**
- If 2 colors specified relative to different white points, they can be related to each other using **chromatic adaptation transformation (CAT)**

XYZ Scaling



- Simplest chromatic adaptation method is XYZ scaling
- Basically, color coordinates multiplied by ratios of corresponding white point coordinates

$$X_2 = X_1 \cdot \frac{X_{W2}}{X_{W1}} \qquad Y_2 = Y_1 \cdot \frac{Y_{W2}}{Y_{W1}} \qquad Z_2 = Z_1 \cdot \frac{Z_{W2}}{Z_{W1}}$$

- For example to convert colors from system based on white point $\mathbf{W_1}$ = **D65** to system relative to $\mathbf{W_2}$ = **D50**

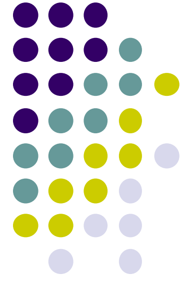
$$X_{50} = X_{65} \cdot \frac{X_{\mathbf{D50}}}{X_{\mathbf{D65}}} = X_{65} \cdot \frac{0.964296}{0.950456} = X_{65} \cdot 1.01456\,,$$

$$Y_{50} = Y_{65} \cdot \frac{Y_{\mathbf{D50}}}{Y_{\mathbf{D65}}} = Y_{65} \cdot \frac{1.000000}{1.000000} = Y_{65}\,,$$

$$Z_{50} = Z_{65} \cdot \frac{Z_{\mathbf{D50}}}{Z_{\mathbf{D65}}} = Z_{65} \cdot \frac{0.825105}{1.088754} = Z_{65} \cdot 0.757843$$

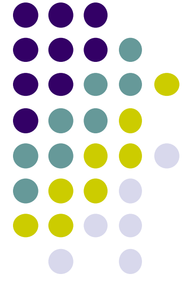
- Another alternative is **Bradford adaptation**

Colorimetric Support in Java



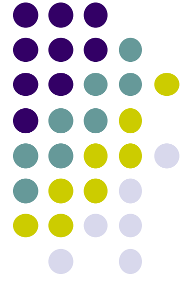
- sRGB is standard color space in Java
- Components of color objects and RGB color images are color corrected

Statistics of Color Images



- **Task:** Determine how many unique colors in a given image
- **Approach 1:** Create histogram, count frequency of each color
  - Not efficient since for 24-bit image, there are 2^{24} = 16,777,216 colors
- **Approach 2:** Put pixels in array, then sort
  - Similar colors next to each other
  - We can then count the number of **transitions between neighboring colors**
  - Much more efficient than histogram approach if many repeated colors

Counting Colors in RGB Image

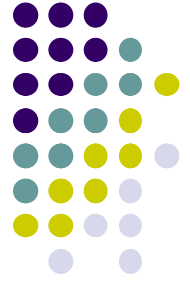


```
1  import ij.process.ColorProcessor;
2  import java.util.Arrays;
3
4  public class ColorStatistics {
5
6    static int countColors (ColorProcessor cp) {
7      // duplicate pixel array and sort
8      int[] pixels = ((int[]) cp.getPixels()).clone();
9      Arrays.sort(pixels);
10
11     int k = 1; // image contains at least one color
12     for (int i = 0; i < pixels.length-1; i++) {
13       if (pixels[i] != pixels[i+1])
14         k = k + 1;
15     }
16     return k;
17   }
18
19 } // end of class ColorStatistics
```

Create copy of 1D RGB array, Then sort

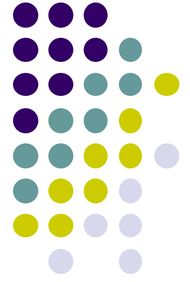
Count transitions between contiguous color blocks

Color Histograms



- When applied to object of type **ColorProcessor**, built-in ImageJ method **getHistogram( )**, simply counts intensity of corresponding gray values
- **Another alternative:** compute individual intensity histograms of 3 color channels
  - **Downside:** does not give information about actual colors in image
- **A good alternative:** 2D color histograms

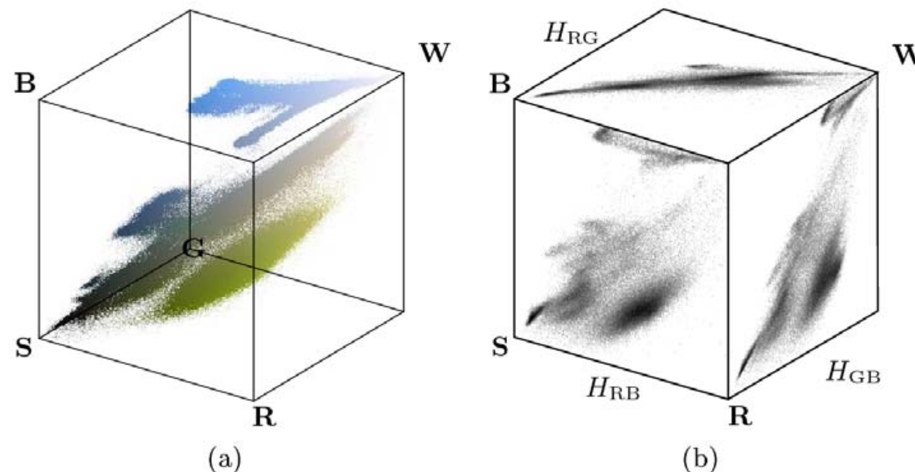
2D Color Histograms



- Define:

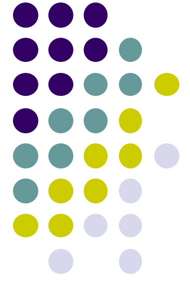
$$H_{\mathrm{RG}}(r,g) \leftarrow \text{number of pixels with } I_{\mathrm{RGB}}(u,v) = (r,g,*)$$
$$H_{\mathrm{RB}}(r,b) \leftarrow \text{number of pixels with } I_{\mathrm{RGB}}(u,v) = (r,*,b)$$
$$H_{\mathrm{GB}}(g,b) \leftarrow \text{number of pixels with } I_{\mathrm{RGB}}(u,v) = (*,g,b)$$

- **Note:** * denotes arbitrary component value
- Resulting 2D histograms are independent of original image size, easily visualized



(a) (b)

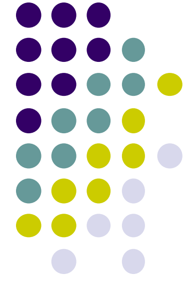
Method for computing 2D Histogram

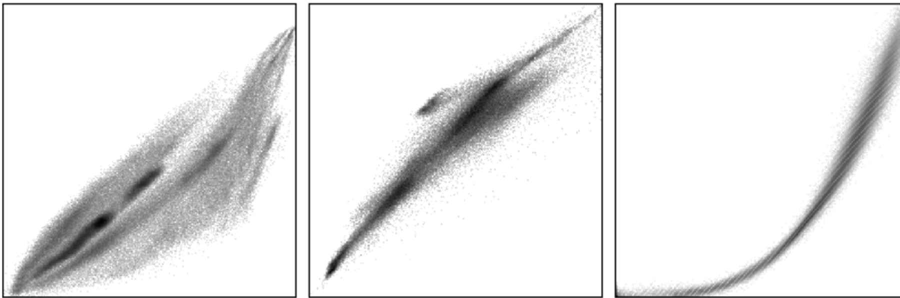


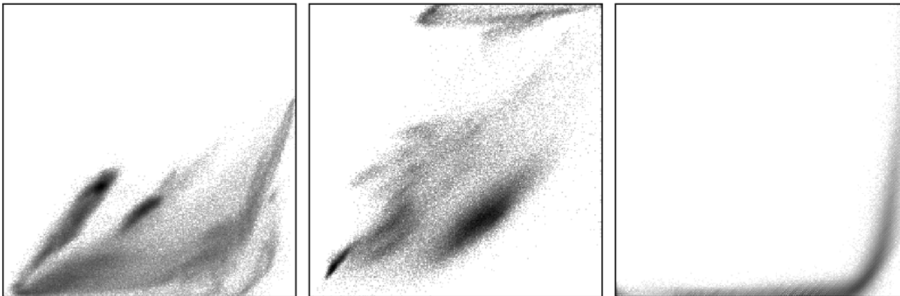
```
1   static int[][] get2dHistogram
2                 (ColorProcessor cp, int c1, int c2) {
3     // c1, c2: R = 0, G = 1, B = 2
4     int[] RGB = new int[3];
5     int[][] H = new int[256][256]; // histogram array H[c1][c2]
6
7     for (int v = 0; v < cp.getHeight(); v++) {
8       for (int u = 0; u < cp.getWidth(); u++) {
9         cp.getPixel(u, v, RGB);
10        int i = RGB[c1];
11        int j = RGB[c2];
12        // increment corresponding histogram cell
13        H[j][i]++; // i runs horizontal, j runs vertical
14      }
15    }
16    return H;
17  }
```

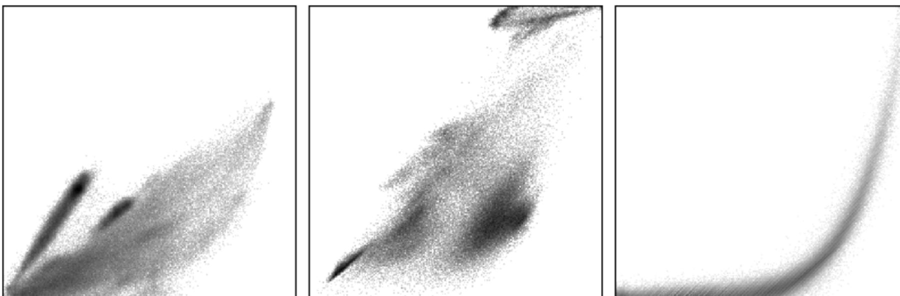
Color components (Histogram axes) Specified by c1 and c2

Color distribution H returned in 2D array



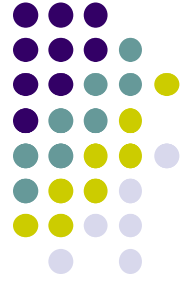






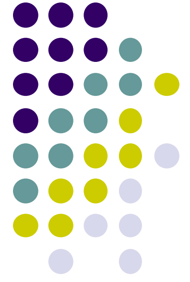
Example: Combined Color Histogram

Color Quantization



- True color can be quite large in actual description
- Sometimes need to reduce size
- **Examples:**
  - Convert 24-bit TIFF to 8-bit TIFF
  - Take a true-color description from database and convert to web image format
- Replace true-color with “best match” from smaller subset
- Quantization algorithms:
  - Uniform quantization
  - Popularity algorithm
  - Median-cut algorithm
  - Octree algorithm

Scalar Color Quantization



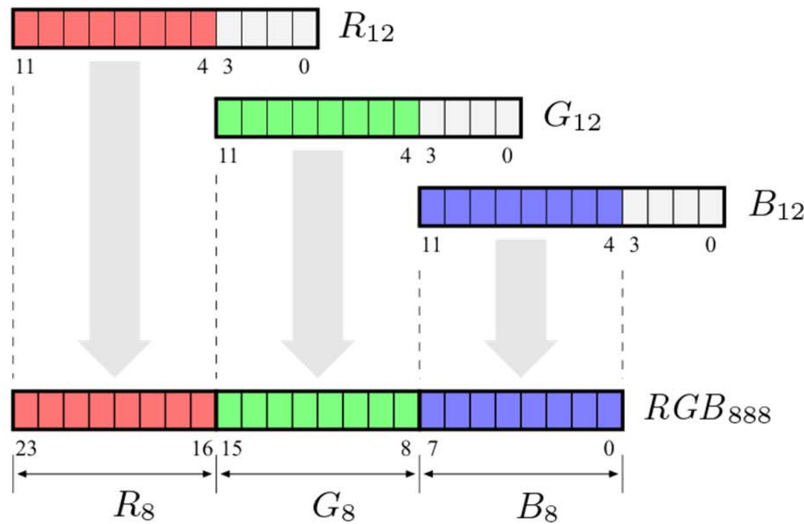
- Convert each component of each original RGB value independently from range [0,…, m-1] to [0,….n-1]

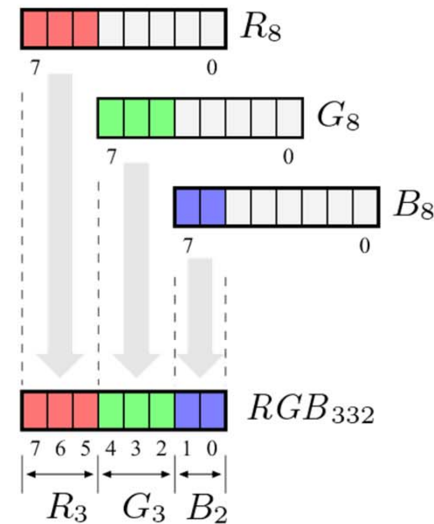
$$c_i' \leftarrow \left\lfloor c_i \cdot \frac{n}{m} \right\rfloor$$

- **Example:** Convert color image with 3x12-bit components (m=4096) to RGB image with 3x8-bit components (n=256)
  - Multiply each component in orginal color by n/m = 256/4096 = 1/16
  - Result is then truncated
- Sometimes m and n not same for all components
- E.g. 3:3:2 packing = 3 bits for red, 3 bits for green and 2 bits for blue

Scalar Quantization

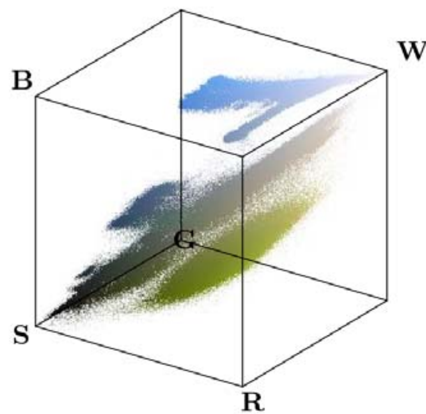
Quantization of 3x12-bit to 3x8-bit colors

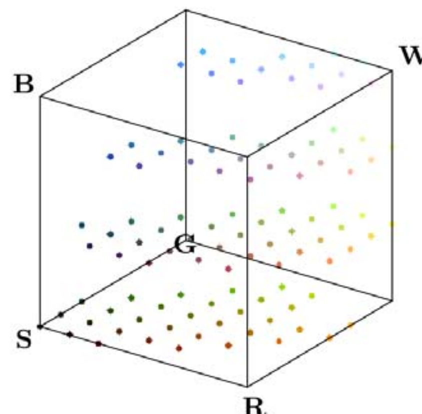




Quantization of 3x8-bit to 3:3:2-packed 8-bit colors

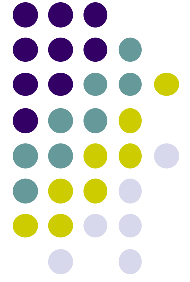
Distribution of 226,321 colors before scalar 3:3:2 quantization



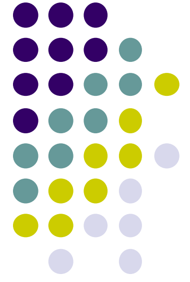


256 colors after scalar 3:3:2 quantization

Vector Quantization



- Does not treat individual components separately
- Each color of pixel treated as single entity
- **Goal:**
  1. To find set of *n* representative color vectors
  2. Replace each original color by one of the new color vectors
- *n* is usually pre-determined
- Resulting deviation should be minimal
- Turns out to be an expensive global optimization problem
- Following methods thus calculate local optima
  - Populousity algorithm
  - Median-cut algorithm

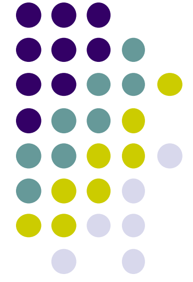


Populosity Algorithm

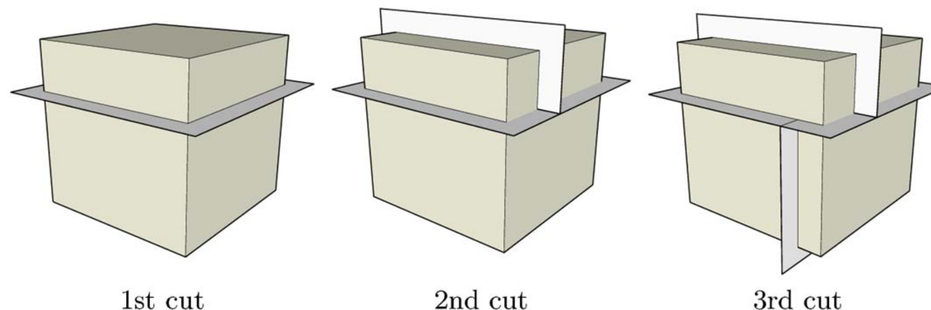
- Selects *n* most frequently occurring colors in image as representative set of color vectors
- General algorithm:
  - Sort image colors into array,
  - Populate histogram as before
  - pick *n* most frequently occurring colors as representative set
  - For each original color, replace with closest color in representative set

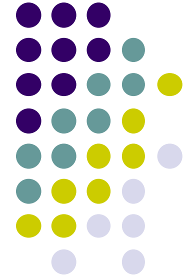
    Note: closest color also shortest distance in color space
- Algorithm performs well as long as colors not scattered a lot

Median-Cut Algorithm



- Considered classical method for color quantization
- Implemented in many applications and in ImageJ
- Algorithm:
  - Compute color histogram of original image
  - Recursively divide RGB color space till number of boxes equal to desired number of representative colors is reached
  - At each step of recursion, box with most pixels is split at median of the longest of its 3 axes so that half pixels left in each subbox
  - In the last step, the mean color of all pixels in each subbox is computed and used as the representative color (each contained pixel is replaced by this mean)



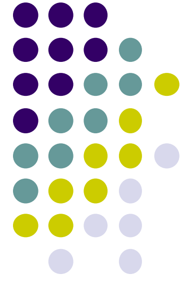


Median-Cut Algorithm Part 1 of 3

1: MedianCut$(I, K_{\max})$
   I: color image, $K_{\max}$: max. number of quantized colors
   Returns a new quantized image with at most $K_{\max}$ colors.
2: $\mathcal{C}_R \leftarrow$ FindRepresentativeColors$(I, K_{\max})$
3: **return** QuantizeImage$(I, \mathcal{C}_R)$

4: FindRepresentativeColors$(I, K_{\max})$
5: Determine $\mathcal{C} = \{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots \boldsymbol{c}_K\}$, the set of K distinct colors in I, where each color instance $\boldsymbol{c}_i$ is a tuple $\langle \mathsf{red}, \mathsf{grn}, \mathsf{blu}, \mathsf{cnt} \rangle$ consisting of the RGB color components (red, grn, blu) and the number of pixels (cnt) for that particular color.
6: **if** $|\mathcal{C}| \leq K_{\max}$ **then**
7: $\mathcal{C}_R \leftarrow \mathcal{C}$.
8: **else**
   Create a color box $\boldsymbol{b}_0$ at level 0 that contains all image colors $\mathcal{C}$ and make it the initial element in the set of color boxes $\mathcal{B}$:
9: $\boldsymbol{b}_0 \leftarrow$ CreateColorBox$(C, 0)$ ▷ see Alg. 12.2
10: $\mathcal{B} \leftarrow \{\boldsymbol{b}_0\}$ ▷ initial set of color boxes
11: $k \leftarrow 1$
12: $done \leftarrow \mathsf{false}$
13: **while** $k < N_{\max}$ **and** not $done$ **do**
14: $\boldsymbol{b} \leftarrow$ FindBoxToSplit$(\mathcal{B})$ ▷ see Alg. 12.2
15: **if** $\boldsymbol{b} \neq \mathsf{nil}$ **then**
16: $\langle \boldsymbol{b}_1, \boldsymbol{b}_2 \rangle \leftarrow$ SplitBox$(\boldsymbol{b})$ ▷ see Alg. 12.2
17: $\mathcal{B} \leftarrow \mathcal{B} - \{\boldsymbol{b}\}$ ▷ remove $\boldsymbol{b}$ from $\mathcal{B}$
18: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\boldsymbol{b}_1, \boldsymbol{b}_2\}$ ▷ insert $\boldsymbol{b}_1$, $\boldsymbol{b}_2$ into $\mathcal{B}$
19: $k \leftarrow k + 1$
20: **else** ▷ no more boxes to split
21: $done \leftarrow \mathsf{true}$
   Determine the average color inside each color box in set $\mathcal{B}$:
22: $\mathcal{C}_R \leftarrow \{\boldsymbol{c}_j = \text{AverageColors}(\boldsymbol{b}_j) \mid \boldsymbol{b}_j \in \mathcal{B}\}$ ▷ see Alg. 12.3
23: **return** $\mathcal{C}_R$.

24: QuantizeImage$(I, \mathcal{C}_R)$
   Returns a new image with color pixels from I replaced by their closest representative colors in $\mathcal{C}_R$:
25: Create a new image I' the same size as I.
26: **for** all (u, v) **do**
27: Find the color $\boldsymbol{c} \in \mathcal{C}_R$ that is "closest" to $I(u, v)$ (e. g., using the Euclidean distance in RGB space).
28: $I'(u, v) \leftarrow \boldsymbol{c}$
29: **return** I'.



Median-Cut Algorithm Part 2 of 3

1: $\textsc{CreateColorBox}(\mathcal{C}, m)$

Creates and returns a new color box containing the colors $\mathcal{C}$. A color box $\boldsymbol{b}$ is a tuple $\langle \mathsf{colors}, \mathsf{level}, \mathsf{rmin}, \mathsf{rmax}, \mathsf{gmin}, \mathsf{gmax}, \mathsf{bmin}, \mathsf{bmax} \rangle$, where colors is the set of image colors represented by the box, level denotes the split-level, and $\mathsf{rmin}, \ldots \mathsf{bmax}$ describe the color boundaries of the box in RGB space.

2: Find the RGB extrema of all colors in this box:

$$\left.\begin{aligned} r_{\min} &\leftarrow \min\ \mathsf{red}(\boldsymbol{c}) \\ r_{\max} &\leftarrow \max\ \mathsf{red}(\boldsymbol{c}) \\ g_{\min} &\leftarrow \min\ \mathsf{grn}(\boldsymbol{c}) \\ g_{\max} &\leftarrow \max\ \mathsf{grn}(\boldsymbol{c}) \\ b_{\min} &\leftarrow \min\ \mathsf{blu}(\boldsymbol{c}) \\ b_{\max} &\leftarrow \max\ \mathsf{blu}(\boldsymbol{c}) \end{aligned}\right\} \text{ for all colors } \boldsymbol{c} \in \mathcal{C}$$

3: Create a new color box $\boldsymbol{b}$:

$\boldsymbol{b} \leftarrow \langle \mathcal{C}, m, r_{\min}, r_{\max}, g_{\min}, g_{\max}, b_{\min}, b_{\max} \rangle$

4: **return** $\boldsymbol{b}$.

5: $\textsc{FindBoxToSplit}(\mathcal{B})$

Searches the set of boxes $\mathcal{B}$ for a box to split and returns this box, or nil if no splittable box can be found.

Let $\mathcal{B}_s$ be the set of all color boxes that can be split (i.e., contain at least 2 different colors):

6: $\mathcal{B}_s \leftarrow \{\, \boldsymbol{b} \mid \boldsymbol{b} \in \mathcal{B} \wedge |\mathsf{colors}(\boldsymbol{b})| \geq 2 \}$

7: **if** $\mathcal{B}_s = \{\}$ **then** ▷ no splittable box was found

8: **return** nil.

9: **else**

10: Select a box $\boldsymbol{b}_x \in \mathcal{B}_s$, such that $\mathsf{level}(\boldsymbol{b}_x)$ is a minimum.

11: **return** $\boldsymbol{b}_x$.

12: $\textsc{SplitBox}(\boldsymbol{b})$

Splits the color box $\boldsymbol{b}$ at the median plane perpendicular to its longest dimension and returns a pair of new color boxes.

13: $m \leftarrow \mathsf{level}(\boldsymbol{b})$

14: $d \leftarrow \textsc{FindMaxBoxDimension}(\boldsymbol{b})$ ▷ see Alg. 12.3

15: $\mathcal{C} \leftarrow \mathsf{colors}(\boldsymbol{b})$

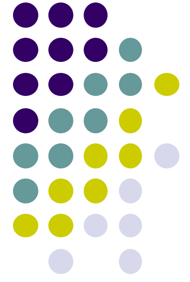
16: From all color samples in $\mathcal{C}$ determine x_{med} as the **median** of the color distribution along dimension d.

17: Partition the set $\mathcal{C}$ into two disjoint sets $\mathcal{C}_1$ and $\mathcal{C}_2$ by splitting at x_{med} along dimension d.

18: $\boldsymbol{b}_1 \leftarrow \textsc{CreateColorBox}(\mathcal{C}_1, m+1)$

19: $\boldsymbol{b}_2 \leftarrow \textsc{CreateColorBox}(\mathcal{C}_2, m+1)$

20: **return** $\langle \boldsymbol{b}_1, \boldsymbol{b}_2 \rangle$.

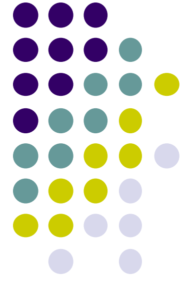


Median-Cut Algorithm Part 3 of 3

1: AVERAGECOLORS($\boldsymbol{b}$)
   Returns the average color $\boldsymbol{c}_{\text{avg}}$ for the pixels represented by the color box $\boldsymbol{b}$.
2: $\mathcal{C} \leftarrow \mathsf{colors}(\boldsymbol{b})$
3: $n \leftarrow 0, \quad r_{\text{sum}} \leftarrow 0, \quad g_{\text{sum}} \leftarrow 0, \quad b_{\text{sum}} \leftarrow 0$
4: **for** all $\boldsymbol{c} \in \mathcal{C}$ **do**
5: $\quad k \leftarrow \mathsf{cnt}(\boldsymbol{c})$
6: $\quad n \leftarrow n + k$
7: $\quad r_{\text{sum}} \leftarrow r_{\text{sum}} + k \cdot \mathsf{red}(\boldsymbol{c})$
8: $\quad g_{\text{sum}} \leftarrow g_{\text{sum}} + k \cdot \mathsf{grn}(\boldsymbol{c})$
9: $\quad b_{\text{sum}} \leftarrow b_{\text{sum}} + k \cdot \mathsf{blu}(\boldsymbol{c})$
10: $r_{\text{avg}} \leftarrow \frac{1}{n} \cdot r_{\text{sum}}, \quad g_{\text{avg}} \leftarrow \frac{1}{n} \cdot g_{\text{sum}}, \quad b_{\text{avg}} \leftarrow \frac{1}{n} \cdot b_{\text{sum}}$
11: $\boldsymbol{c}_{\text{avg}} \leftarrow \langle r_{\text{avg}}, g_{\text{avg}}, b_{\text{avg}} \rangle$
12: **return** $\boldsymbol{c}_{\text{avg}}$.

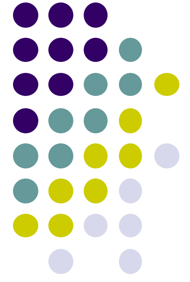
13: FINDMAXBOXDIMENSION($\boldsymbol{b}$)
   Returns the largest dimension of the color box $\boldsymbol{b}$ (i. e., Red, Green, or Blue).
14: $\text{size}_r = \mathsf{rmax}(\boldsymbol{b}) - \mathsf{rmin}(\boldsymbol{b})$
15: $\text{size}_g = \mathsf{gmax}(\boldsymbol{b}) - \mathsf{gmin}(\boldsymbol{b})$
16: $\text{size}_b = \mathsf{bmax}(\boldsymbol{b}) - \mathsf{bmin}(\boldsymbol{b})$
17: $\text{size}_{\text{max}} = \max(\text{size}_r, \text{size}_g, \text{size}_b)$
18: **if** $\text{size}_{\text{max}} = \text{size}_r$ **then**
19: $\quad$ **return** Red.
20: **else if** $\text{size}_{\text{max}} = \text{size}_g$ **then**
21: $\quad$ **return** Green.
22: **else**
23: $\quad$ **return** Blue.

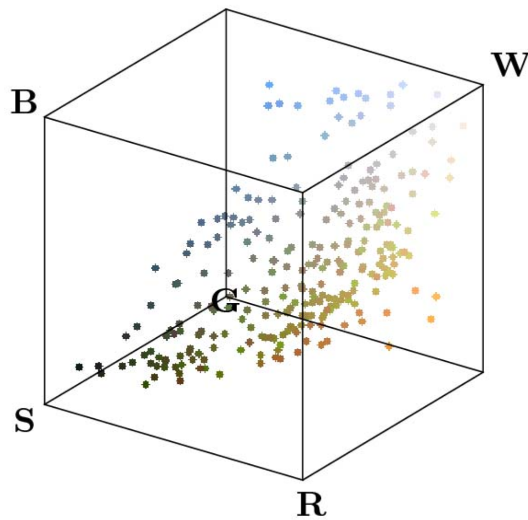
Octree & Other Quantization Algorithms



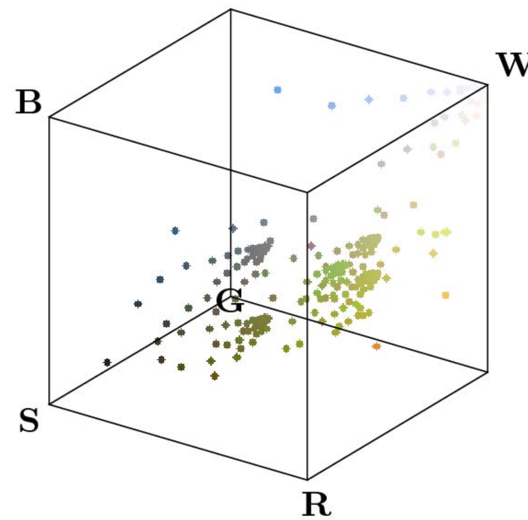
- Octree:
  - Similar to median cut
  - Hierarchical structure: each cube may be split into **8 subcubes**
- **Other approaches:**
  - Use 10% of pixels that are randomly selected as representative
  - Statistical and clustering methods. E.g. k-means

Color Distribution after Median Cut





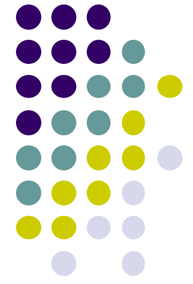
(a)



(b)

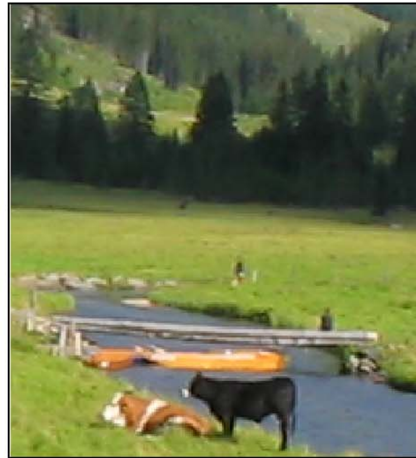
Color distribution after reducing original 226,321 colors to 256 using median cut

Color distribution after reducing original 226,321 colors to 256 using octree

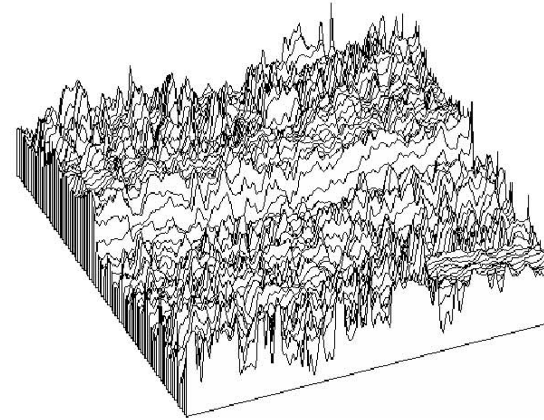


Comparison of Quantization Errors

Original Image



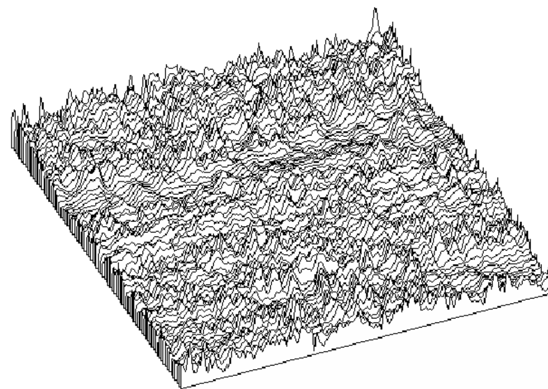
(a) Detail



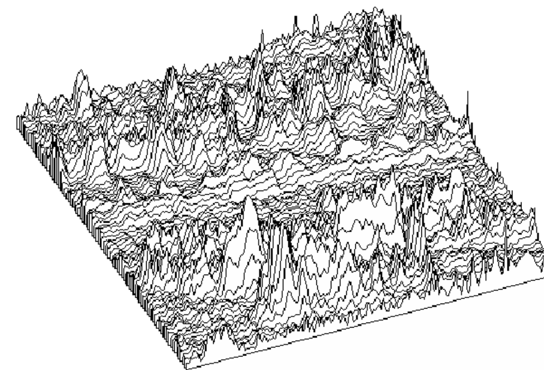
(b) 3:3:2

Distance between original and quantized color for scalar 3:3:2 packing

Median Cut



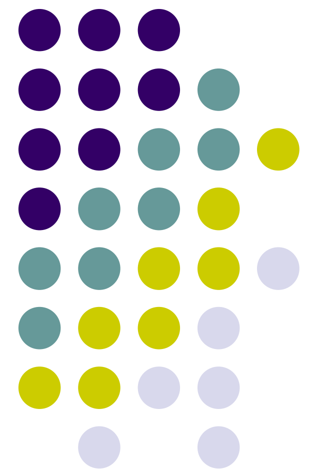
(c) Median-Cut



(d) Octree

Octree

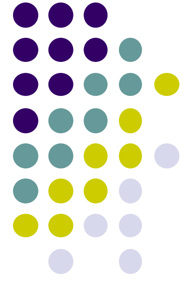
Digital Image Processing (CS/ECE 545) Lecture 9: Color Images (Part 2) & Introduction to Spectral Techniques (Fourier Transform, DFT, DCT)



Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)



Fourier Transform

- **Main idea:** Any periodic function can be decomposed into a summation of sines and cosines

Complex function → $y = f(x)$

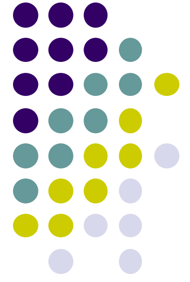
Sine function 1 → $y = \sin x$

Sine function 2 → $y = \frac{1}{3}\sin 2x$

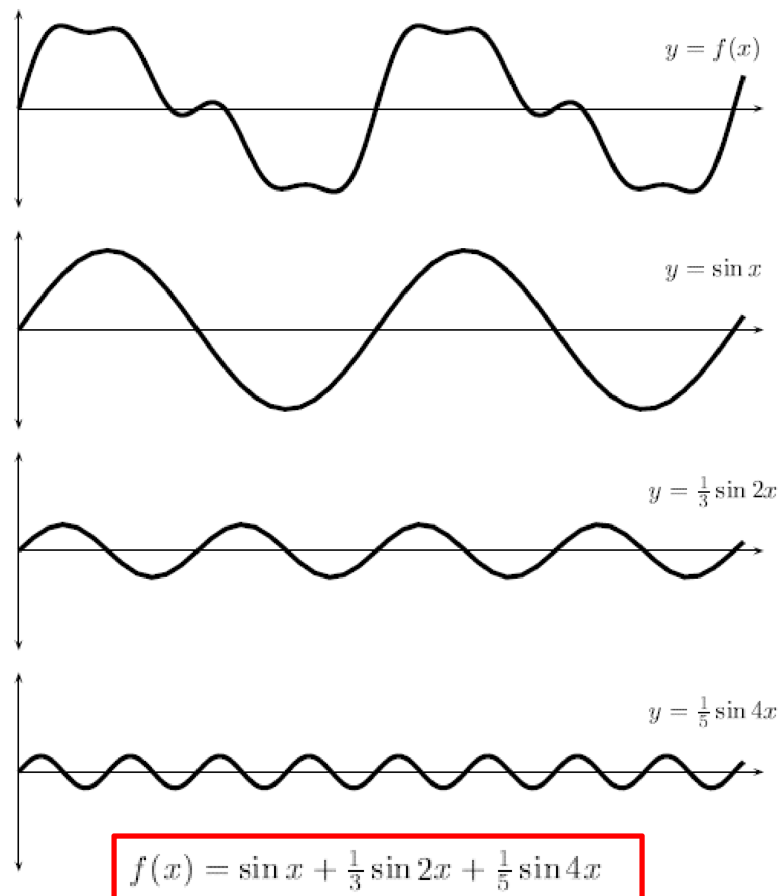
Sine function 3 → $y = \frac{1}{5}\sin 4x$

Complex function expressed as sum of sines → $f(x) = \sin x + \frac{1}{3}\sin 2x + \frac{1}{5}\sin 4x$

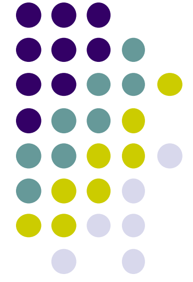
Fourier Transform: Why?

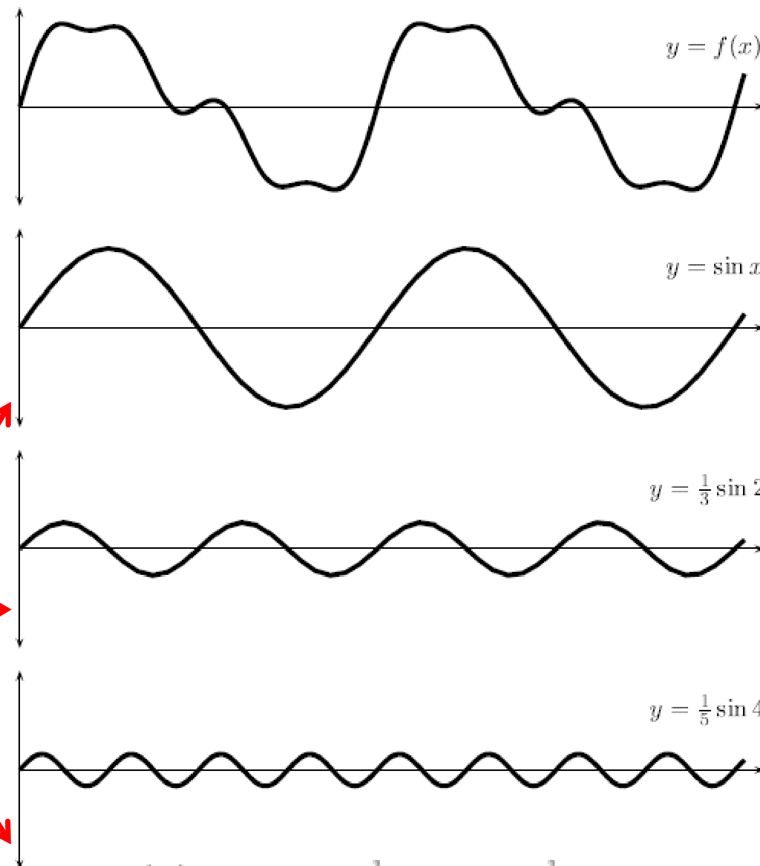


- Mathematially easier to analyze effects of transmission medium, noise, etc on simple sine functions, then add to get effect on complex signal



Fourier Transform: Some Observations





Observation 2: Frequencies of sines are multiples of each other (called harmonics)

Frequency = 1x

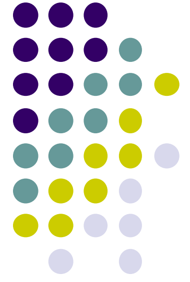
Frequency = 2x

Frequency = 4x

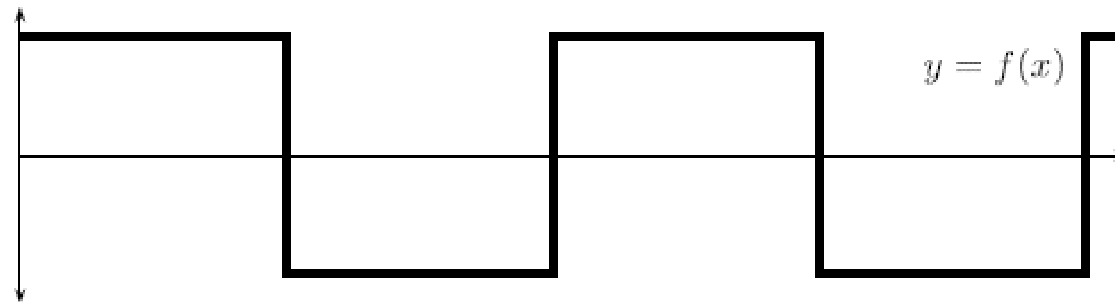
Observation 1: The sines have different frequencies (not same)

Observation 3: Different amounts of different sines added together (e.g. 1/3, 1/5, etc)

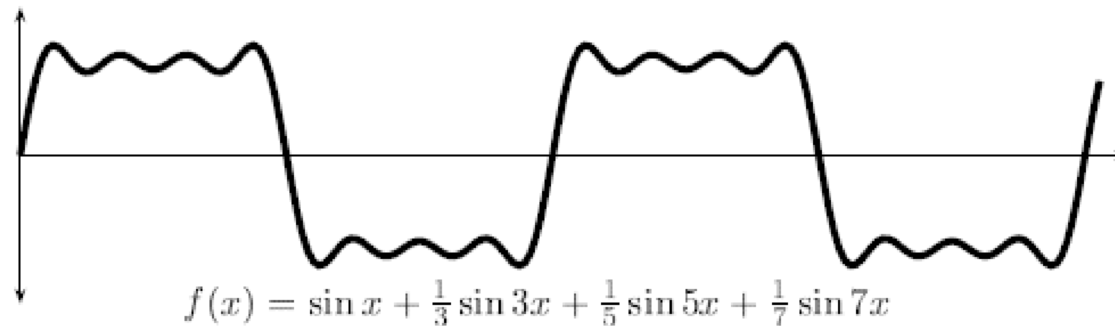
Fourier Transform: Another Example



Square wave

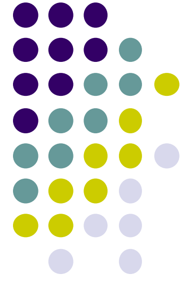


Approximation
Using sines



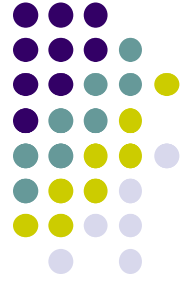
Observation 4: The sine terms go to infinity.
The more sines we add, the closer the
approximation of the original.

Who is Fourier?

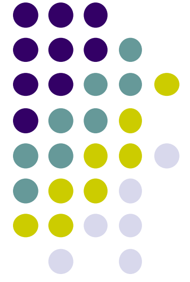


- French mathematician and physicist
- Lived 1768 - 1830

Spectral Techniques



- Technique for representation and analysis of signals in frequency domain including audio, images, video
- How to decompose signal into summation of a series of sine and cosine functions (also called harmonic functions)
- Spectral techniques can improve efficiency of image processing
- Some image processing effects, concepts, techniques easier in frequency domain
- Includes **fourier transform**, **discrete fourier transform** and **discrete cosine transform**



Sine and Cosine Functions: A Review

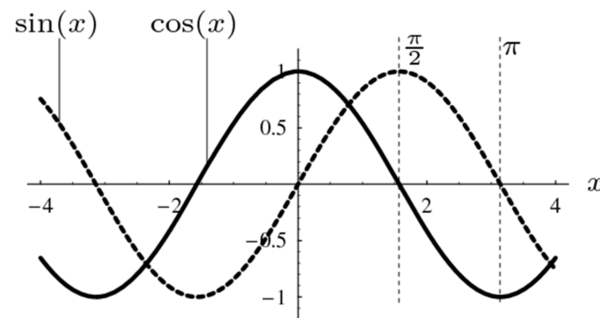
- Cosine function

$$f(x) = \cos(x)$$

- A function is periodic if

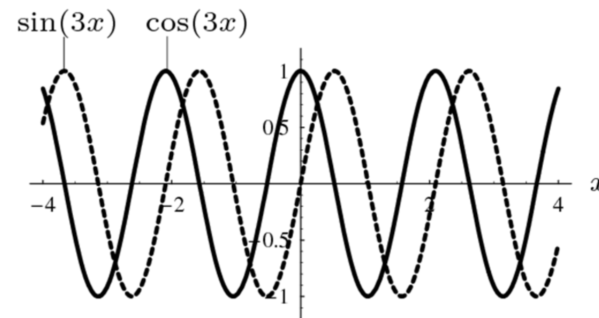
$$\cos(x) = \cos(x + 2\pi) = \cos(x + 4\pi) = \cdots = \cos(x + k2\pi)$$

- Sines and cosines at different frequencies



(a)

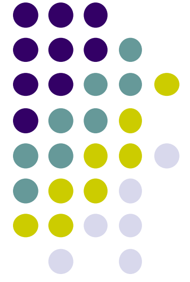
Frequency = 1x



(b)

Frequency = 3x

Sines and Cosines: A Review



- Relationship between period *T,* frequency *f* and angular velocity *ω*

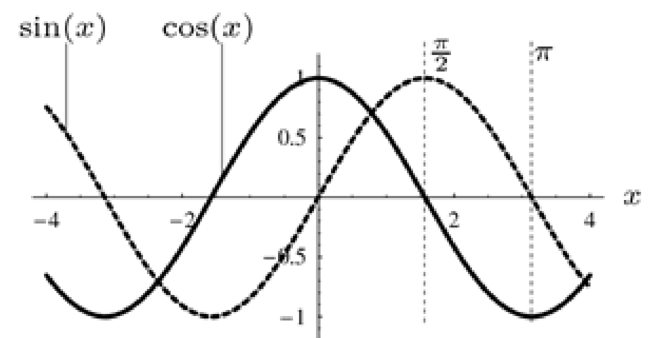
$$f = \frac{1}{T} = \frac{\omega}{2\pi} \quad \text{or} \quad \omega = 2\pi f$$

- Shifting phase of a cosine function

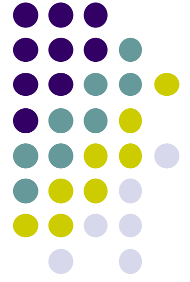
$$\cos(x) \rightarrow \cos(x - \varphi)$$

- Sine = cos shifted to right by 90 degrees

$$\sin(\omega x) = \cos\left(\omega x - \tfrac{\pi}{2}\right)$$



Sines and Cosines: A Review

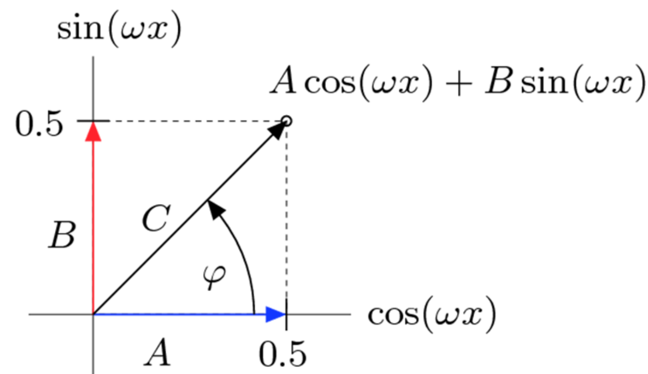


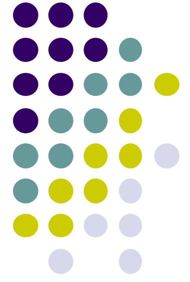
- Adding a sine and cosine with same frequencies but arbitrary amplitudes *A* and *B* creates another sinusoid

$$A \cdot \cos(\omega x) + B \cdot \sin(\omega x) = C \cdot \cos(\omega x - \varphi)$$

- Amplitude and phase angle of *C* are

$$C = \sqrt{A^2 + B^2} \quad \text{and} \quad \varphi = \tan^{-1}\left(\tfrac{B}{A}\right)$$





Sines and Cosines: A Review

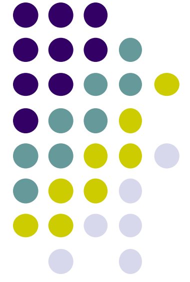
- Euler's complex number notation

$$z = e^{\mathrm{i}\theta} = \cos(\theta) + \mathrm{i} \cdot \sin(\theta)$$

- Easy to combine sines and cosines of same frequency

$$e^{\mathrm{i}\theta} = e^{\mathrm{i}\omega x} = \cos(\omega x) + \mathrm{i} \cdot \sin(\omega x)$$

Fourier Series of Periodic Functions



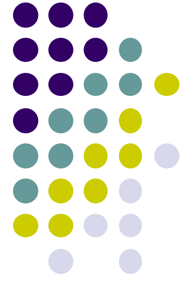
- (Almost) any periodic function *g(x)* with fundamental frequency ω_0 can be described as a sum of sinusoids

$$g(x) = \sum_{k=0}^{\infty} [A_k \cos(k\omega_0 x) + B_k \sin(k\omega_0 x)]$$

Infinite sum of **Cosines** **Sines**

- This infinite sum is called a **Fourier Series**
- Summed sines and cosines are multiples of the fundamental frequency (harmonics)
- $\boldsymbol{A_k}$ and $\boldsymbol{B_k}$ called **Fourier coefficients**
  - Not known initially but derived from original function *g(x)* during **Fourier analysis**

Fourier Integral



- For non-periodic functions similar ideas yield the **Fourier Integral** (integration of densely packed sines and cosines

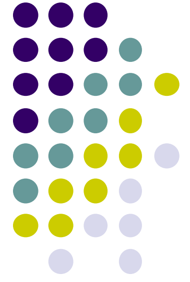
$$g(x) = \int_0^\infty A_\omega \cos(\omega x) + B_\omega \sin(\omega x) \, \mathrm{d}\omega$$

where coefficients can be found as

$$A_\omega = A(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \cos(\omega x) \, \mathrm{d}x$$

$$B_\omega = B(\omega) = \frac{1}{\pi} \int_{-\infty}^{\infty} g(x) \cdot \sin(\omega x) \, \mathrm{d}x$$

Fourier Transform



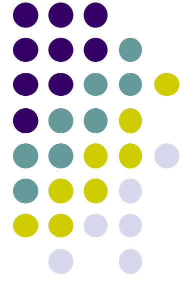
- **Fourier Transform:** Transition of function *g(x)* to its Fourier spectrum *G(ω)*

$$G(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot \left[\cos(\omega x) - \mathrm{i} \cdot \sin(\omega x)\right] \, \mathrm{d}x$$
$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(x) \cdot e^{-\mathrm{i}\omega x} \, \mathrm{d}x.$$

- **Inverse Fourier Transform:** Reconstruction of original function *g(x)* from its Fourier spectrum *G(ω)*

$$g(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot \left[\cos(\omega x) + \mathrm{i} \cdot \sin(\omega x)\right] \, \mathrm{d}\omega$$
$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} G(\omega) \cdot e^{\mathrm{i}\omega x} \, \mathrm{d}\omega.$$

References



- Wilhelm Burger and Mark J. Burge, Digital Image Processing, Springer, 2008
- University of Utah, CS 4640: Image Processing Basics, Spring 2012
- Rutgers University, CS 334, Introduction to Imaging and Multimedia, Fall 2012
- Gonzales and Woods, Digital Image Processing (3rd edition), Prentice Hall
- Computer Graphics Using OpenGL 2nd edition by F.S Hill Jr, chapter 12