VK Multimedia Information Systems

Mathias Lux, mlux@itec.uni-klu.ac.at

Tuesday, 10 am

Content Based Image Retrieval

- Motivation & Semantic Gap
- Perception
- Color Based Features
- Texture Based Features

Motivation

Lots of good reasons …

- Visual information overload
  - Devices (cameras, mobile phones, etc.)
  - Communication (email, mo-blogs, etc.)
- Metadata not available
  - Time consuming
  - No automation


Question: What is so special ‘bout Mona Lisa’s smile?

Semantic Gap

- Defined as
  - Inability of automatic understanding
  - Gap between high- and low-level features / metadata
- Actually hard task for humans also

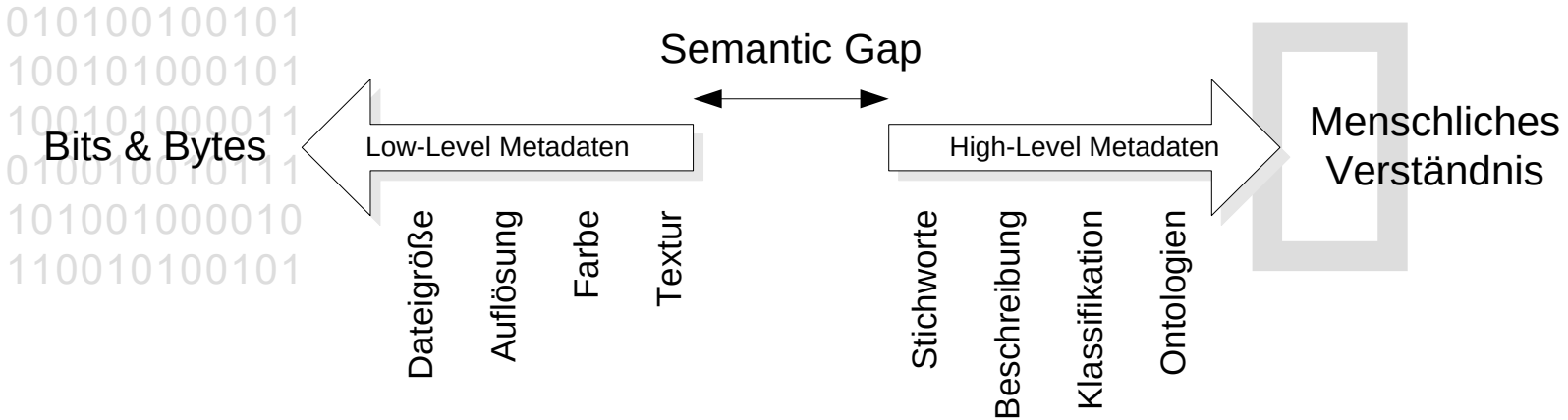


Image Similarity

Are these two images similar?

Semantic Gap (1)

- General Definition: Santini & Jain (1998)



Applications

- Home User & Entertainment
  - Find picture of / from / at ....
  - Search & browse personal digital library
- Graphics & Design
  - Find picture representing something (Color in CD/CI, feeling, etc.)
- Medical Applications
  - Find images for diagnosis, documentation

Applications

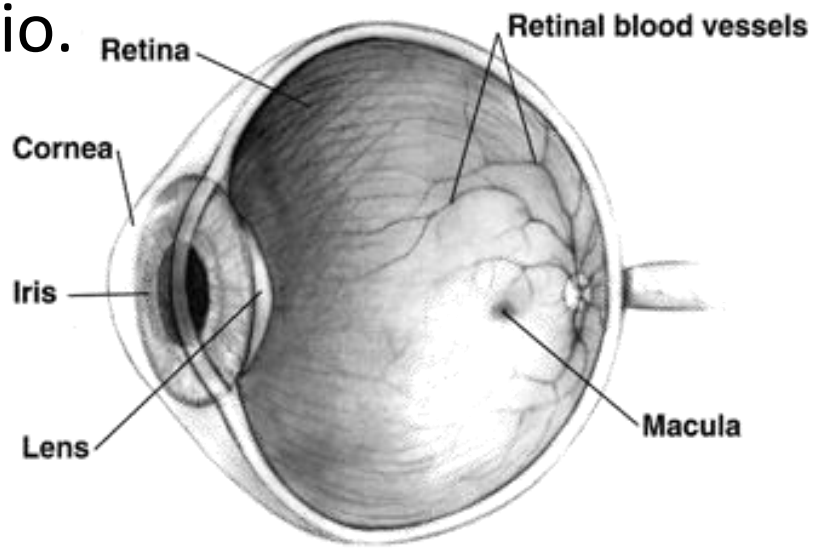
- Biology
  - Finding similar animals, insects & plants
- Weather forecasting
  - Finding similar weather conditions
- Advertisement
  - Find similar products

Content Based Image Retrieval

- Motivation & Semantic Gap
- Perception
- Color Based Features
- Texture Based Features

Perception

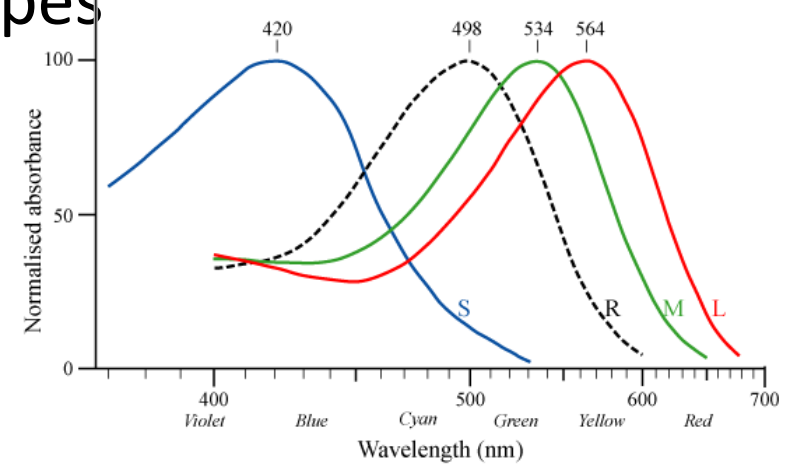
- The **eye** as instrument of perception
- Sensory capabilities
  - Cones (bright light): 6-7 Mio.
  - Rods (dim-light): 75-150 Mio.
  - Brain ‘corrects’ vision
    - e.g. blind spot



Color & Color Spaces

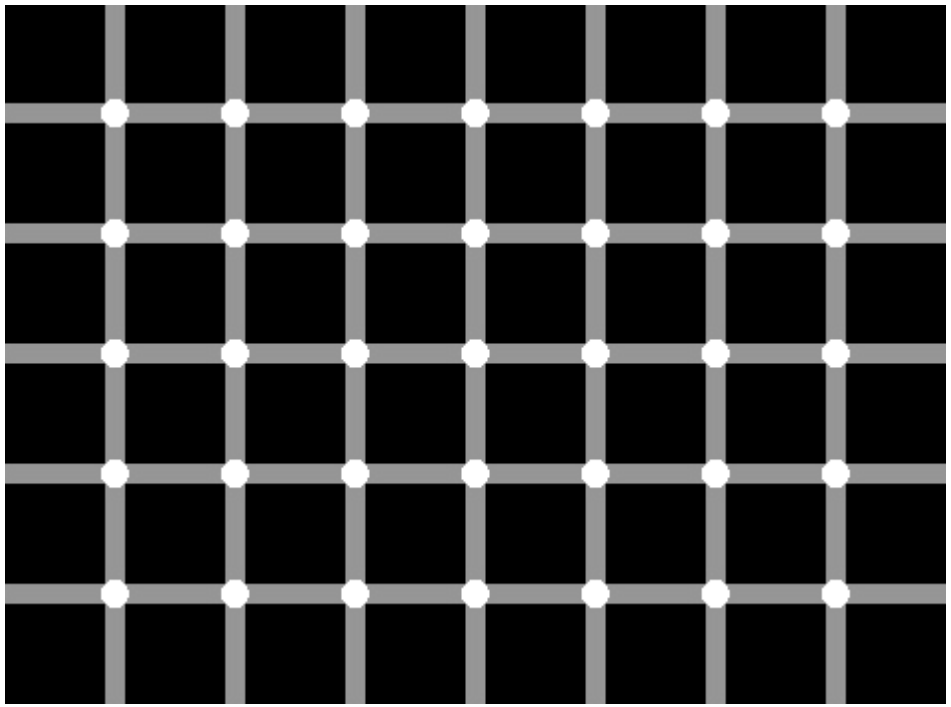
S-, M- and L-cones: Blue, green and red

- RGB based on these three colors
- CIE models perception better
  - Responsiveness of cone types
  - Number of cones / types
  - etc.



The human eye ...

- Count the black dots on the image:



The human eye ...

- Rabbit or duck?

The human eye ...

- Anamorphic illusions

See e.g. http://users.skynet.be/J.Beever/pave.htm

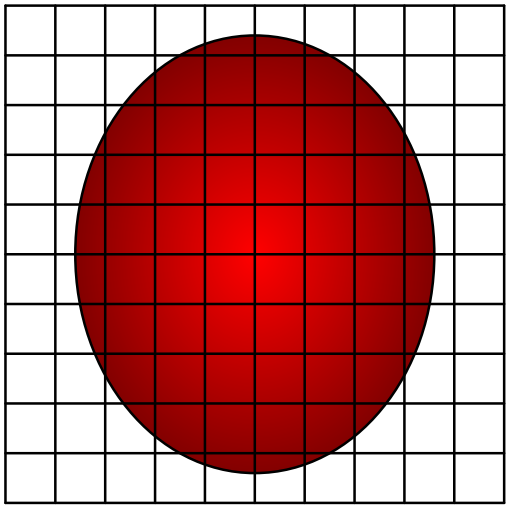
Anamorphic Illusions (Julian Beever)

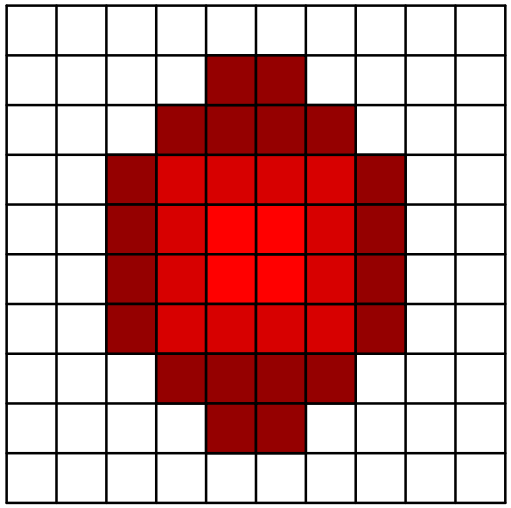
What are (digital) images?

- An Image is
  - Created by a set of photons
    - With different frequency
    - Moving from different sources
    - Along different vectors
  - A representation of sensor unit activation
    - Activated by the set of photons
- Storing an image
  - Based on the set of photons ???

Sampling & Quantization

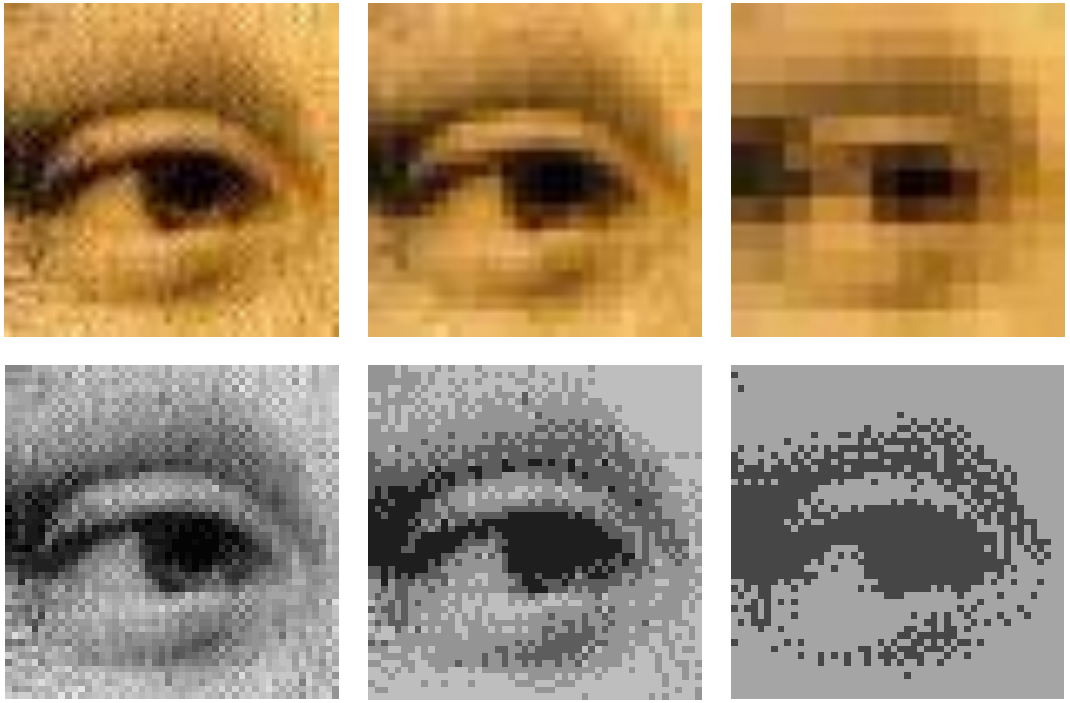
- Capturing continuous images on sensors
  - Sampling: Continuous to matrix
  - Quantization: Continuous color to value





Sampling & Quantization

- Size of a captured image:
  - # of samples (width*heigth) * # of colors



Content Based Image Retrieval

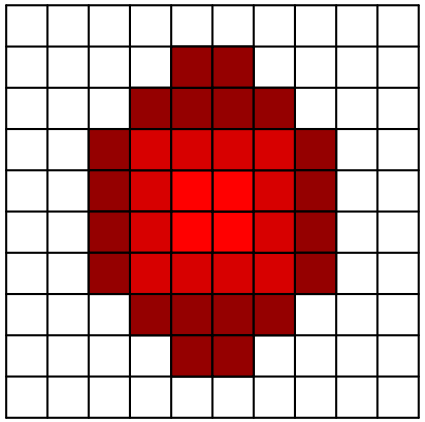
- Motivation & Semantic Gap
- Perception
- Color Based Features
- Texture Based Features

Accessing Pixel Data in Java

```
// opening an image file
BufferedImage image = ImageIO.read(new File("ferrari-01.jpg"));
// pixel data is in the raster
WritableRaster raster = image.getRaster();
// create an object for the pixel data:
int[] pixel = new int[3];
// access pixel data:
for (int x = 0; x < raster.getWidth(); x++) {
     for (int y = 0; y < raster.getHeight(); y++) {
          // access pixel data here ...
          raster.getPixel(x, y, pixel);
     }
}
```

Histogram

- Count the use of different colors
- Algorithm:
  - Allocate int array h with dim = # of colors
  - Visit next pixel -> it has color with index i
  - Increment h[i]
  - IF pixels left THEN goto line 2
- Example: 4 colors, 10*10 pixels
  - histogram: [4, 12, 20, 64]



Histogram

```
int[] histogram = new int[32];
for (int x = 0; x < raster.getWidth(); x++) {
   for (int y = 0; y < raster.getHeight(); y++) {
        histogram[quantize(raster.getPixel(x, y, pixel))]++;
   }
}
```

Luminance Histogram

```
public static int quantize(int[] px) {
    double tmp = 0.2126*px[0] + 0.7152*px[1]+0.0722*px[2];
    return (int) Math.floor(tmp/8d);
}
```

- Quantizing grey values to 32 bins

Luminance Histogram

23457	74872
51486	40664
39312	37941
28169	31067
22580	29121
26710	36275
35984	38909
57674	42283
69471	37490
60497	31190
64573	25132
44798	21917
44170	19232
44716	17922
29013	17241
19404	16411
13584	15714
5988	15340
2208	14692
1508	15108
1473	12558
1680	10782
1428	9578
1139	8551
812	8035
651	7363
676	7031
591	6700
517	6699
459	7976
557	8599
2059	25975

Color Histogram

```
public static int quantize(int[] pixel) {
    int pos = (int) Math.round((double) pixel[2] / 85d) +
                   (int) Math.round((double) pixel[1] / 85d) * 4 +
                   (int) Math.round((double) pixel[0] / 85d) * 4 * 4;
    return pos;
}
```

- Quantizing RGB color to 64 bins …
  - Note that this “should” be based on a palette …

Color Histogram

478276	414439
87375	56886
71792	31216
28929	22990
14228	18176
3034	14792
1507	13658
1297	11705
1294	9164
1135	7807
884	6719
767	5849
552	5174
452	4733
405	4342
352	4016
308	3727
340	3543
303	3226
256	2944
247	3055
231	2901
191	2792
171	2771
186	2903
185	3349
168	3145
193	3073
227	3298
284	4026
750	7648
1025	14301

Color Histogram

- Strategies:
  - Quantize if too many colors
  - Normalize histogram (different image sizes)
  - Weight colors according to use case
  - Use (part of) color space according to domain
- Distance / Similarity
  - Assumption: All images have the same colors
  - L_1 or L_2 is quite common

Histogram Similarity

- L1 (Manhattan, city block) distance

$$d_1(\mathbf{p},\mathbf{q}) = \|\mathbf{p}-\mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|,$$

- L2 (Euclidean) distance

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2 + (q_2-p_2)^2 + \cdots + (q_n-p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

- Jensen-Shannon Divergence

$$d_{JSD}\left(H,H'\right) = \sum_{m=1}^{M} H_m \log \frac{2H_m}{H_m + H'_m} + H'_m \log \frac{2H'_m}{H'_m + H_m}$$

Histogram Similarity

```
// --- L1 ---
distance = 0;
for (int i = 0; i < hist2.length; i++) {
    distance += Math.abs(hist2[i] - hist1[i]);
}
System.out.println("L1 " + distance);
// --- L2 ---
distance = 0;
for (int i = 0; i < hist2.length; i++) {
    distance += (hist2[i] - hist1[i]) * (hist2[i] - hist1[i]);
}
distance = (float) Math.sqrt(distance);
System.out.println("L2 " + distance);
// --- JSD ---
distance = 0;
for (int i = 0; i < hist2.length; i++) {
    distance += (float) hist1[i] * Math.log(2f * (float) hist1[i] / ((float) hist1[i] + (float) hist2[i])) +
                (float) hist2[i] * Math.log(2f * (float) hist2[i] / ((float) hist1[i] + (float) hist2[i]));
}
distance = (float) Math.sqrt(distance);
System.out.println("JSD " + distance);
```

Histogram Similarity

- Luminance
  - L1 436406.0
  - L2 77265.8
  - JSD 343.3
- RGB Color
  - L1 82706.0
  - L2 58174.6
  - JSD 281.5

Color Histogram

- Benefits
  - Easy to compute, not depending on pixel order
  - Matches human perception quite well
  - Quantization allows to scale size of histogram
  - Invariant to (lossless) rotation & reflection
- Disadvantages
  - Distribution of colors not taken into account
  - Image scaling changes color

Opponent Histogram

- Transformation of RGB
  - O3 … Intensity
  - O1,O1 … Color

$$\begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix} = \begin{bmatrix} \frac{R-G}{\sqrt{2}} \\ \frac{R+G-2B}{\sqrt{6}} \\ \frac{R+G+B}{\sqrt{3}} \end{bmatrix}$$

Dominant Color

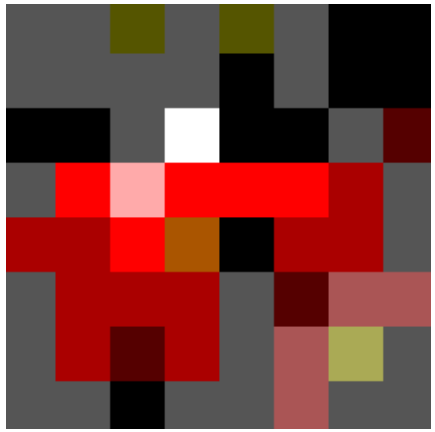
- Reduce histogram to dominant colors
  - e.g. for 64 colors c0-c63:
    - image 1: c12 -> 23%, c33 -> 6%, c2 -> 2%
    - image 2: c11 -> 43%, c2 -> 12%, c54 -> 10%
- Distance function in 2 aspects:
  - Difference in amount (percentage)
  - Difference between colors (c11 vs. c12)
- Further aspects:
  - Diversity and distribution

Dominant Color

- Benefits:
  - Small feature vectors
  - Easily understandable & intuitive
  - Similarity of color pairs (light vs. dark red, etc.)
  - Invariant to rotation & reflection
- Disadvantages
  - Similarity of color pairs no trivial problem
  - Dominant colors might not represent semantics

Color Distribution

- Index dominant color in image segment
  - e.g. 8*8 = 64 image segments
  - feature vector has 64 dimensions
    - One for each segment
  - color index is the entry on segment dimension
    - e.g. 16 colors [2, 0, 3, 3, 8, 4, ...]



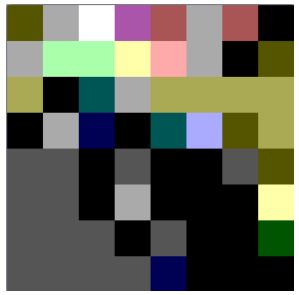
Color Distribution

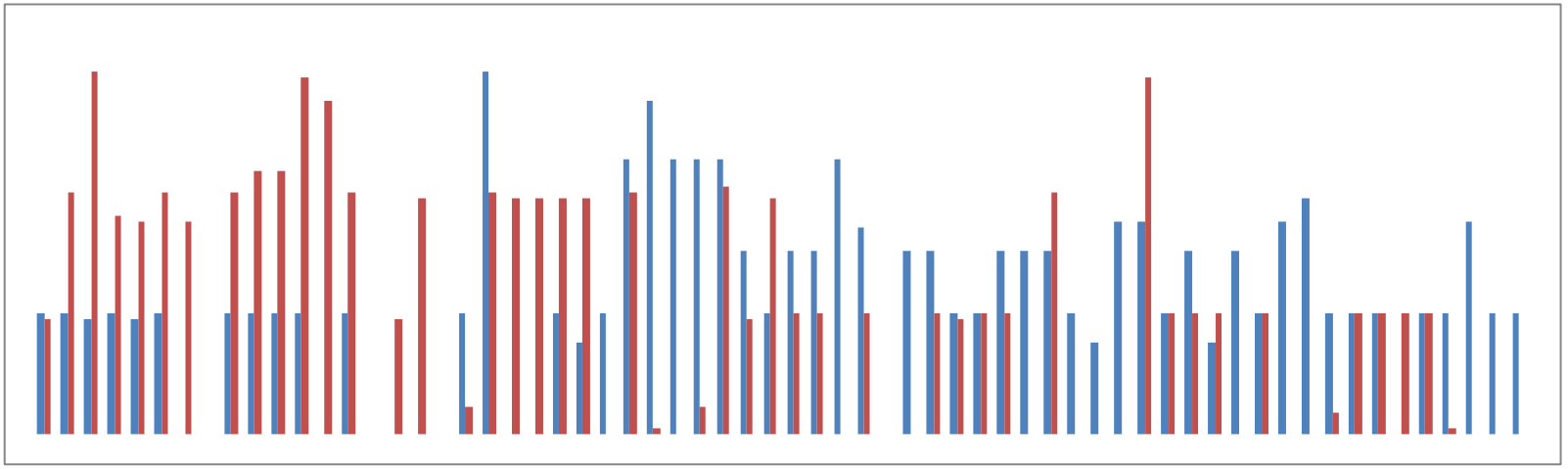
```
BufferedImage image = ImageIO.read(new File("ferrari-01.jpg"));
// created instance of scaled image:
BufferedImage scaled = new
    BufferedImage(8,8,BufferedImage.TYPE_INT_RGB);
// scale image (hardware acceleration)
scaled.getGraphics().drawImage(image,
            0, 0, 8, 8,                                    // destination
            0, 0, image.getWidth(), image.getHeight(),  // source
            null);
```

Color Distribution

```
// quantize and create histogram
WritableRaster raster = scaled.getRaster();
int[] hist = new int[64];
// temporary objects (speed)
int[] pixel = new int[3];
int tmp;
for (int x = 0; x < raster.getWidth(); x++) {
    for (int y = 0; y < raster.getHeight(); y++) {
            raster.getPixel(x,y,pixel);
            tmp = quantize(pixel);
            hist[y*8+x]=tmp;
            // set pixel in scale version for visualization:
            raster.setPixel(x,y,ColorHistogram.palette64[tmp]);
    }
}
```

Color Distribution





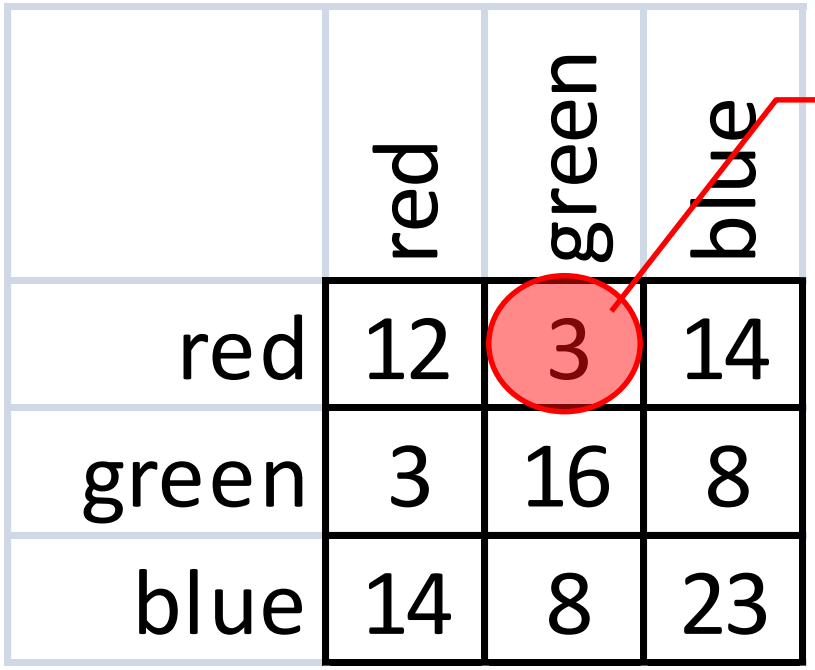
Color Distribution

- Similarity
  - L_1 or L_2 are commonly used
- Benefits
  - Works fine for many scenarios
    - clouds in the sky, portrait photos, etc.
  - Mostly invariant to scaling
- Disadvantages
  - Colors might not represent semantics
  - Find quantization fitting to domain / perception
  - Rotation & reflection are a problem

Color Correlogram

- Histogram on
  - how often **specific colors** occur
  - in the **neighbourhood** of each other
- Histogram size is (# of colors)^2
  - For each color an array of neighboring colors

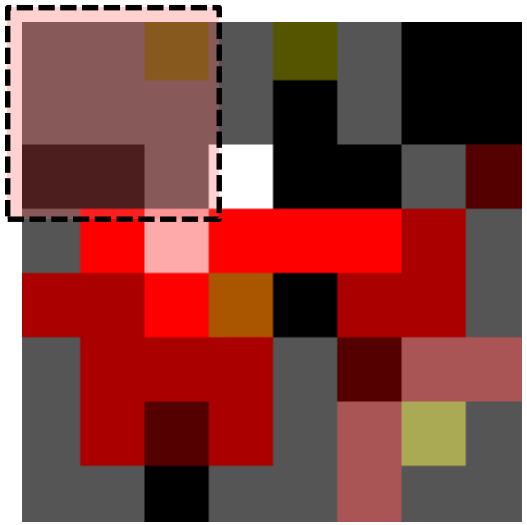
Color Correlogram



	red	green	blue
red	12	3	14
green	3	16	8
blue	14	8	23

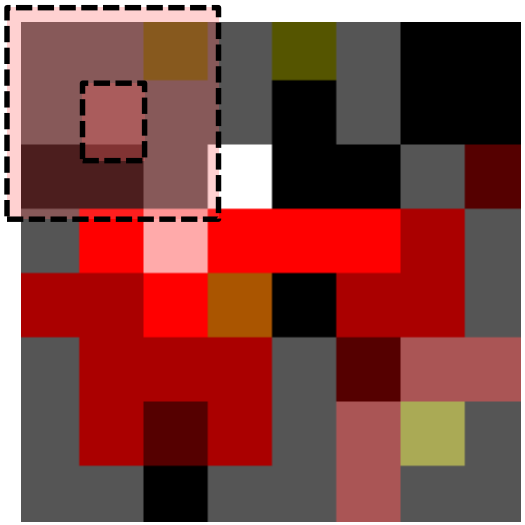
For red pixels we find 3 green pixels all over the image in a neighborhood p.

Color Correlogram



Color Correlogram

- For each window position
- Consider center color
- Add surrounding colors to histogram

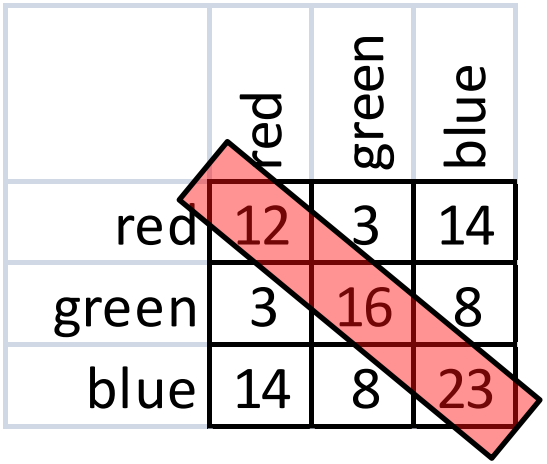


Color Correlogram

- Extraction algorithm
  - Allocate array h[#colors][#colors] all zero
  - Visit next pixel *p*
  - For each pixel *q* in neighborhood of *p*:
    - increment h[color(p)][color(q)]
  - IF pixels left THEN goto line 2
- Algorithm is rather slow
  - Depends on size of neighborhood
  - Typically determined by city block (L1) distance

Color Correlogram

- Auto Color Correlogram
  - Just indexing how often *color(p)* occurs in neighborhood of pixel *p*
  - Simplifies the histogram to size # of colors
  - Measure e.g. how red comes with red etc.



Color Correlogram

- Similarity
  - L_1 or L_2 are commonly used
- Benefits
  - Integrates color as well as distribution
  - Works fine for many scenarios
  - Mostly invariant to rotation & reflection
- Disadvantages
  - Find appropriate neighborhood size
  - Find quantization fitting to domain / perception
  - Rather slow indexing / extraction

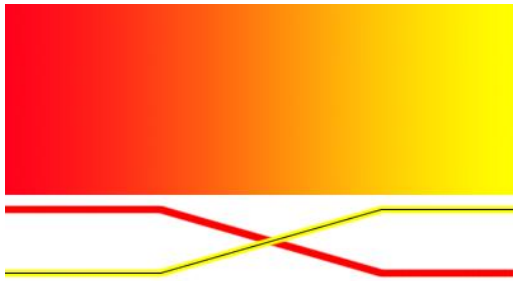
Color Correlogram

- Integrating different pixel features to correlate
  - **Gradient Magnitude** (intensity of change in the direction of maximum change)
  - **Rank** (intensity variation within a neighborhood of a pixel)
  - **Texturedness** (number of pixels exceeding a certain level in a neighborhood)

Fuzzy Color Histogram

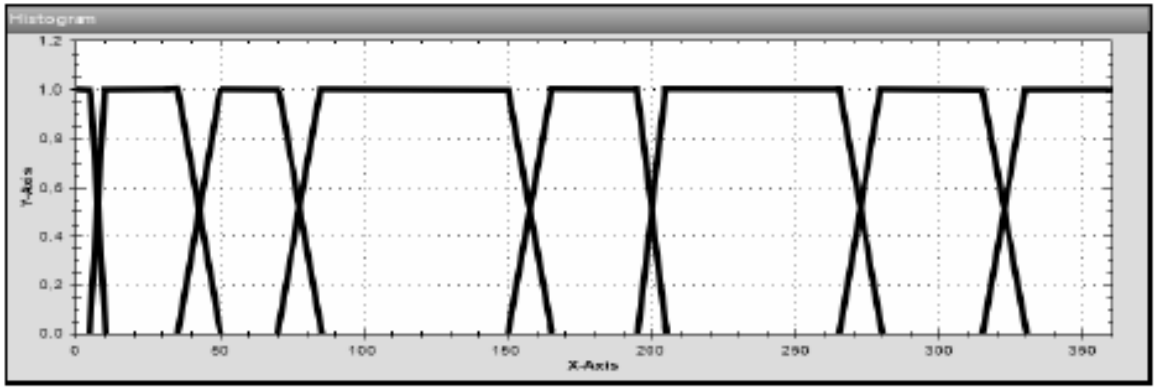
- Binary decision not necessary
- Pixel can be half read, half yellow

- Fuzziness:
  - membership functions $m_i(r,g,b)$
  - For each bin i: $m_i(r,g,b)$ in [0,1]
  - Sum of all m_i = 1.



Fuzzy Color Histogram (FCTH)

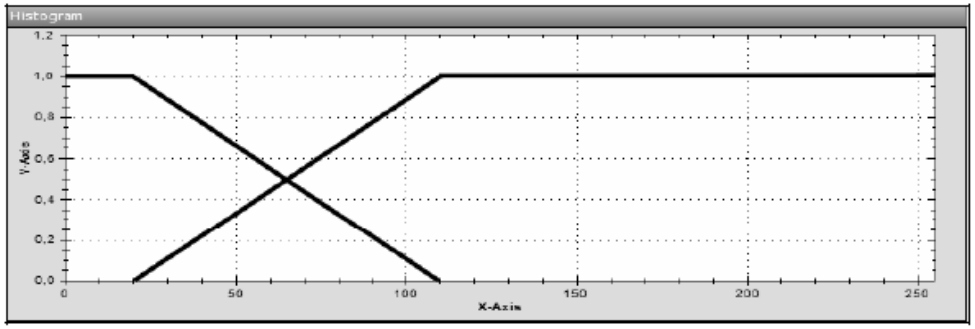
- Fuzzy membership function for HSV in FCTH
  - (0) Red to Orange
  - (1) Orange
  - (2) Yellow
  - (3) Green
  - (4) Cyan
  - (5) Blue
  - (6) Magenta
  - (7) Blue to Red



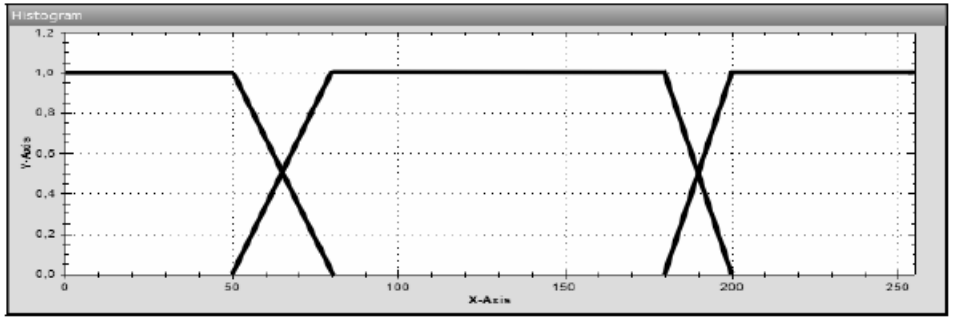
S. A. Chatzichristofis and Y. S. Boutalis, "FCTH: FUZZY COLOR AND TEXTURE HISTOGRAM A LOW LEVEL FEATURE FOR ACCURATE IMAGE RETRIEVAL", WIAMIS IEEE 2008

Fuzzy Color Histogram (FCTH)

- Saturation: white vs. color



- Value: black vs. grey vs. color



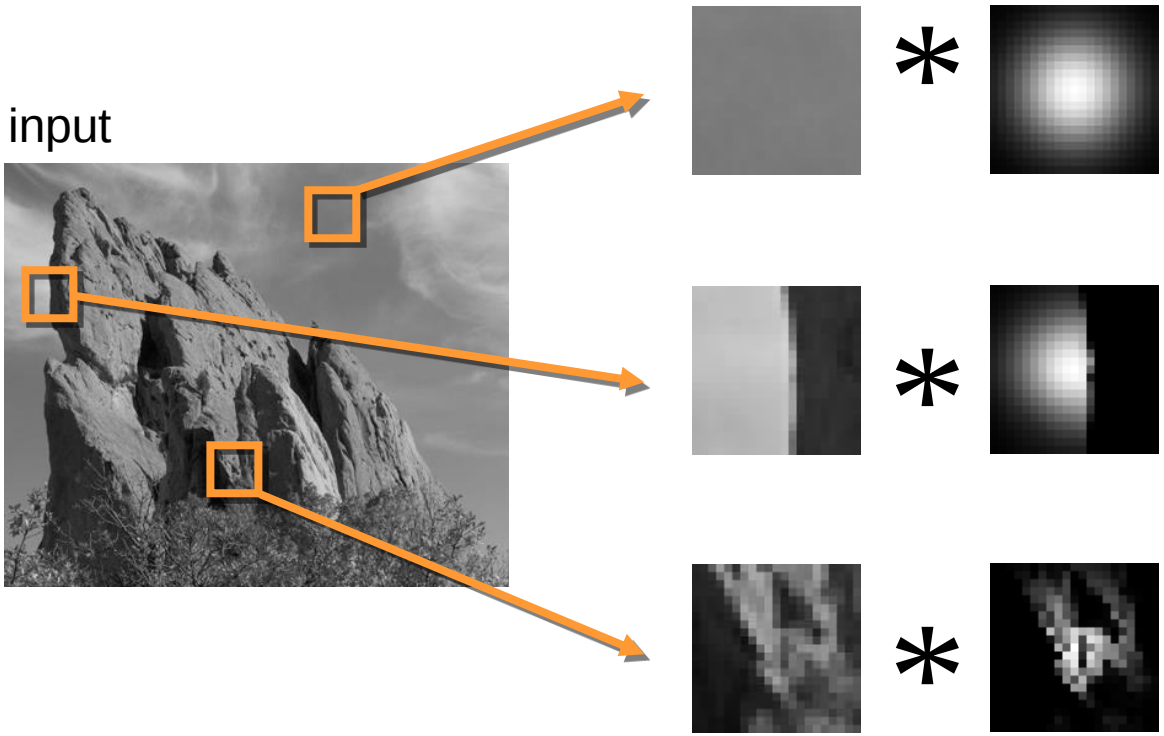
Noise Reduction & Color Quantization

- Preprocessing images for retrieval
  - Removing noise
  - Creating homogeneous patches of color

Bilateral Filtering

- Edge preserving blur filter
- Photoshop „Surface Blur“

Bilateral Filtering



The kernel shape depends on the image content.

src. http://people.csail.mit.edu/sparis/bf_course/

Kuwahara Filter

- Edge preserving blur filter
- Kuwahara window, applied for each pixel

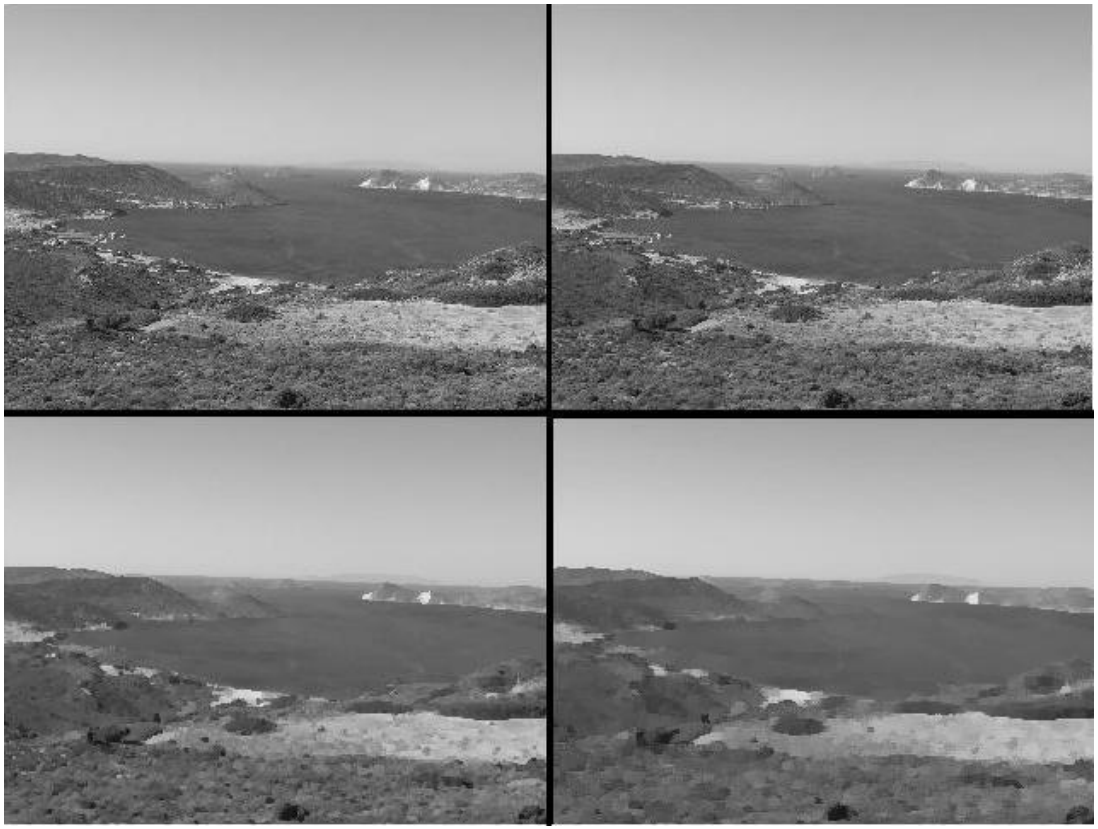
a	a	a/b	b	b
a	a	a/b	b	b
a/c	a/c	a/b/ c/d	b/d	b/d
c	c	c/d	d	d
c	c	c/d	d	d

"Kuwahara" by DinoVgk - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons
https://commons.wikimedia.org/wiki/File:Kuwahara.jpg#/media/File:Kuwahara.jpg

Kuwahara Filter

- Per pixel p for regions a, b, c and d
  - Compute mean and standard deviation
- Select region *n* with smallest standard deviation
- Set p to the mean value of *n*.

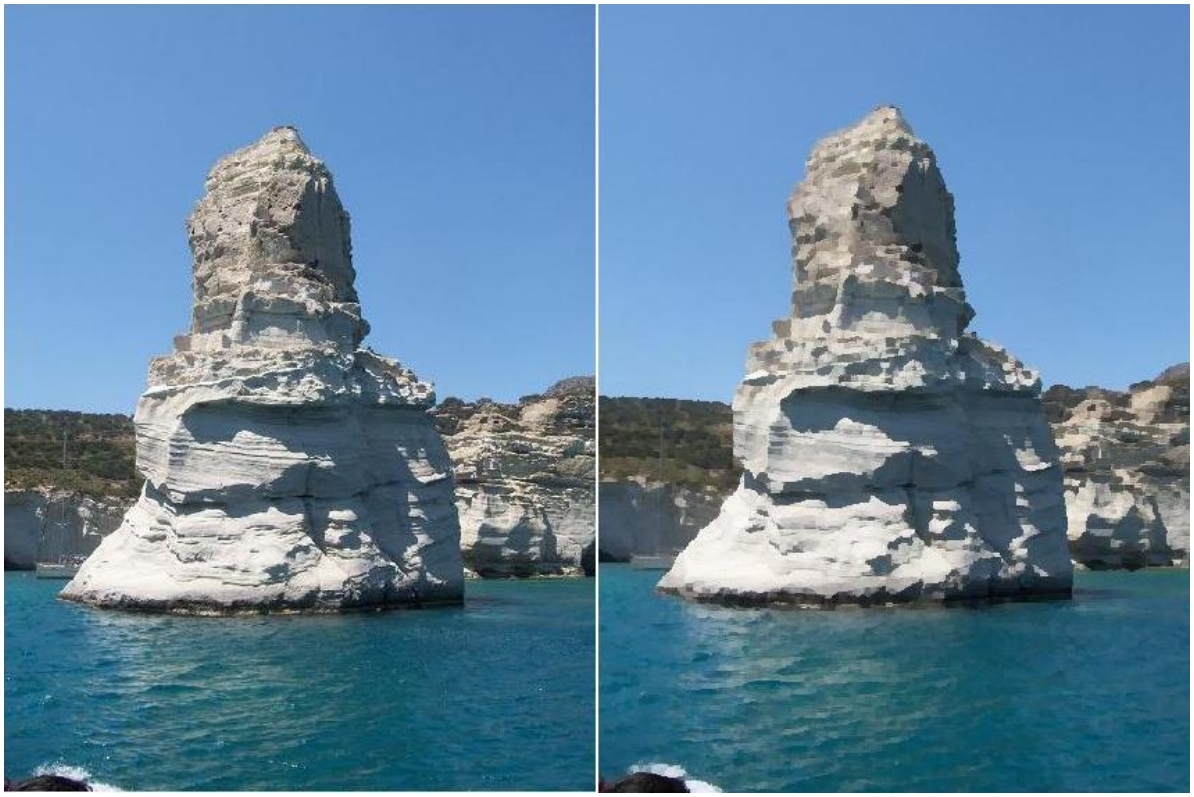
Kuwahara results



Kuwahara Filter for Color Images

- Use HSV as color model
- Just use V (value, brightness) for
  - determination of the standard deviation and
  - decision on which region to use.

Kuwahara color results



Content Based Image Retrieval

- Motivation & Semantic Gap
- Perception
- Color Based Features
- Texture & Shape Features

Texture & Shape Features

- Indexing non color features in image
  - Outlines, edges of regions
  - Overall characteristics like coarseness and regularity

Tamura Features

Tamura & Mori (1978)

- Widely used in CBIR
  - E.g. IBM QBIC
- 6 texture features
  - Coarseness, contrast, directionality
  - Line-likeness, regularity, and roughness
- Good overview is provided in:
  - Thomas Deselaers, “Features for Image Retrieval”, Thesis, RWTH Aachen, Dec. 2003

Tamura Features

Tamura & Mori (1978)

- Coarseness
  - Pixel diversity in neighborhoods
- Contrast
  - Using mean and variance of an image
- Directionality
  - Horizontal and vertical derivatives (like Sobel)

Spatial Filtering

- Methods for *enhancing* the image
- Normally a kernel or filter is used:
  - A matrix which is applied to the image
  - In a linear transformation

Spatial Filtering

194	128	102	197	69
162	68	103	144	115
121	85	57	27	14
24	183	192	239	150
92	93	154	138	170

194	128	102	197	69
162	68	103	144	115
121	85	122	27	14
24	183	192	239	150
92	93	154	138	170

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Spatial Filtering

- This is a simple smoothing kernel
- Other operations
  - Sharpen
  - Gradient
  - …

Edge Detection

- Spatial Filtering with Sobel Kernel

```
// read image
BufferedImage image = ImageIO.read(new File("ferrari-01.jpg"));
// converting to grey
image = convertImageToGrey(image);
// spatial filtering
BufferedImage filteredImage =
        new BufferedImage(image.getWidth(), image.getHeight(), image.getType());
// create Sobel kernel
Kernel kernel = new Kernel(3, 3, new float[] {1,2,1,0,0,0,-1,-2,-1});
ConvolveOp op = new ConvolveOp(kernel);
// filter image
op.filter(image, filteredImage);
```

Convert to grey in Java

```
WritableRaster inRaster = image.getRaster();
BufferedImage result = new BufferedImage(image.getWidth(), image.getHeight(),
    BufferedImage.TYPE_BYTE_GRAY);
WritableRaster outRaster = result.getRaster();
int[] p = new int[3];
float v = 0;
for (int x = 0; x < inRaster.getWidth(); x++) {
    for (int y = 0; y< inRaster.getHeight(); y++) {
            inRaster.getPixel(x, y, p);
            v = Math.round((p[0] + p[1] + p[2])/3f);
            for (int i = 0; i < p.length; i++) {
                    p[i] = (int) v;
            }
            outRaster.setPixel(x,y, p);
    }
}
```

Edge Detection

Edge Detection

- Apply two Kernels
  - with results L_x and L_y
- Then compute the gradient magnitude

$$|\nabla L| = \sqrt{L_x^2 + L_y^2}$$

- Edge direction: $\arctan(L_y / L_x)$

1	2	1
0	0	0
-1	-2	-1

1	0	-1
2	0	-2
1	0	-1

Edge Detection: Sobel Filter

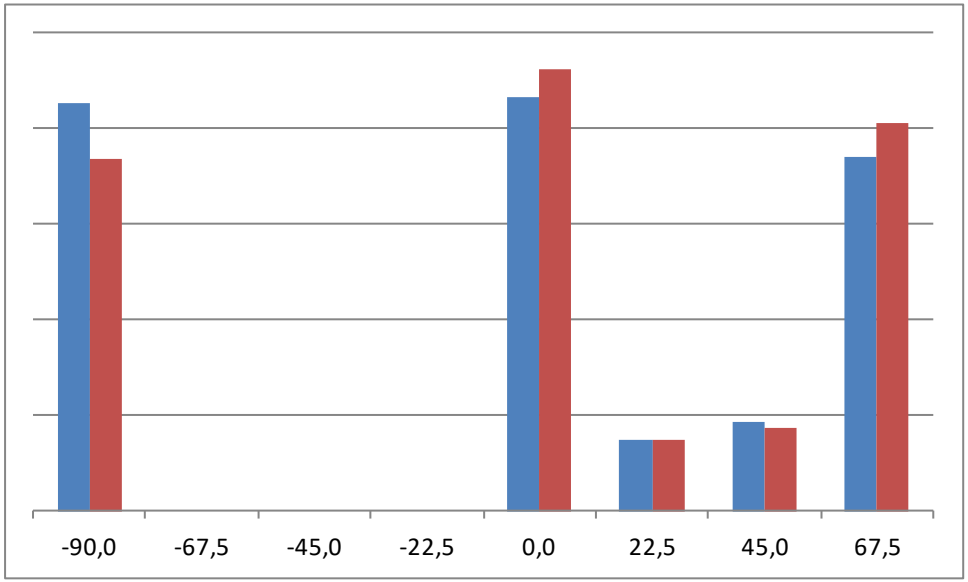
Directionality Histogram

```
// 8-bin directionality histogram ..
int hist[] = new int[8];
for (int i = 0; i < hist.length; i++)
      hist[i] = 0;

for (int x = 0; x < rasterX.getWidth(); x++) {
      for (int y = 0; y < rasterX.getHeight(); y++) {
            rasterX.getPixel(x,y,pixelX);
            rasterY.getPixel(x,y,pixelY);
            // compute directionality histogram:
            double tmpAngle = Math.PI/2 + Math.atan2((double) pixelY[0] / (double) pixelX[0]);
            tmpAngle = Math.floor(8d * tmpAngle / Math.PI);
            tmpAngle = Math.min(7d, tmpAngle);
            hist[((int) tmpAngle)]++;
      }
}
```

Directionality Histogram

- Angle histogram with 8 bins
- [-90, -67.5), [-67.5, -45), …



Difference of Gaussians

- Yet another way to find edges ...

Given a m-channels, n-dimensional image

$$I : \{\mathbb{X} \subseteq \mathbb{R}^n\} \to \{\mathbb{Y} \subseteq \mathbb{R}^m\}$$

The difference of Gaussians (DoG) of the image I is the function

$$\Gamma_{\sigma_1,\sigma_2} : \{\mathbb{X} \subseteq \mathbb{R}^n\} \to \{\mathbb{Z} \subseteq \mathbb{R}\}$$

obtained by subtracting the image I convolved with the Gaussian of variance σ_2^2 from the image I convolved with a Gaussian of narrower variance σ_1^2, with $\sigma_2 > \sigma_1$. In one dimension, Γ is defined as:

$$\Gamma_{\sigma_1,\sigma_2}(x) = I * \frac{1}{\sigma_1\sqrt{2\pi}} e^{-(x^2)/(2\sigma_1^2)} - I * \frac{1}{\sigma_2\sqrt{2\pi}} e^{-(x^2)/(2\sigma_2^2)}.$$

and for the centered two-dimensional case :

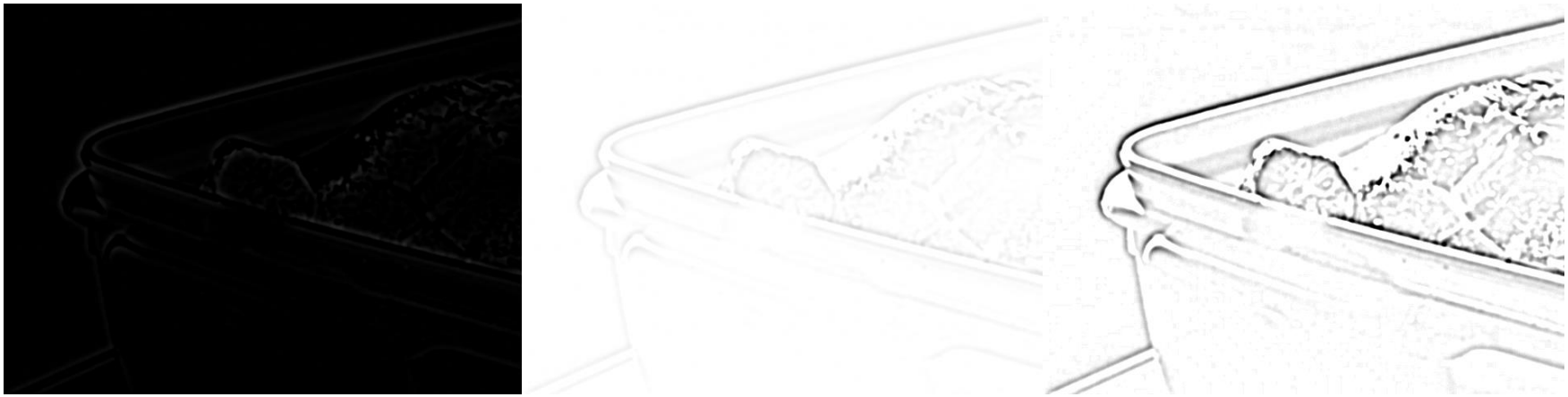
$$\Gamma_{\sigma,K\sigma}(x,y) = I * \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/(2\sigma^2)} - I * \frac{1}{2\pi K^2\sigma^2} e^{-(x^2+y^2)/(2K^2\sigma^2)}$$

src. Wikipedia, https://en.wikipedia.org/wiki/Difference_of_Gaussians

DoG


Original image
Grayscale
Gaussian Blur (4px)
Gaussian Blur (2px)

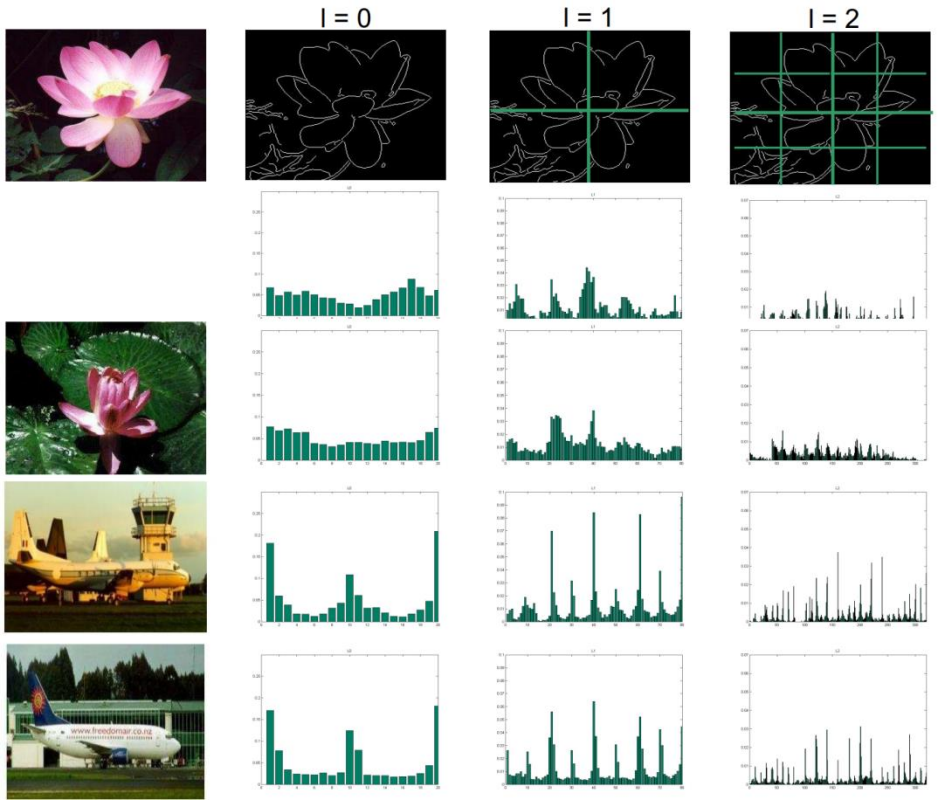
DoG result



result, inverted, contrast enhanced

PHOG
Pyramid Histogram of Gradients

Bosch, Zisserman & Munoz. "Representing shape with a spatial pyramid kernel" CIVR '07

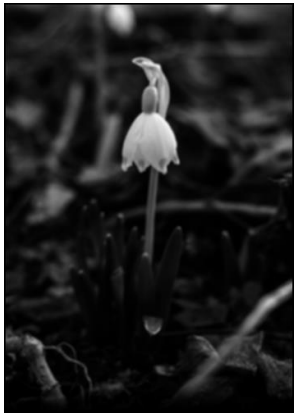


PHOG: Canny Edge Detector

1. Noise reduction
2. Intensity gradient
3. Non-maximum supression
4. Tracing edges

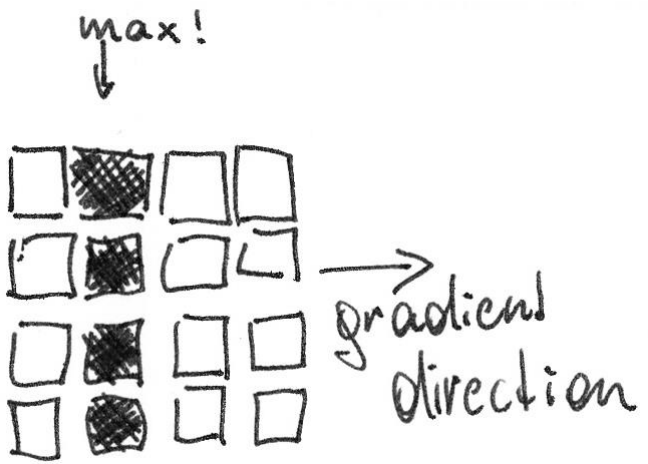
Noise Reduction & Gradient

- Convert to gray
- Run Gaussian blur filter
- Run Sobel filter

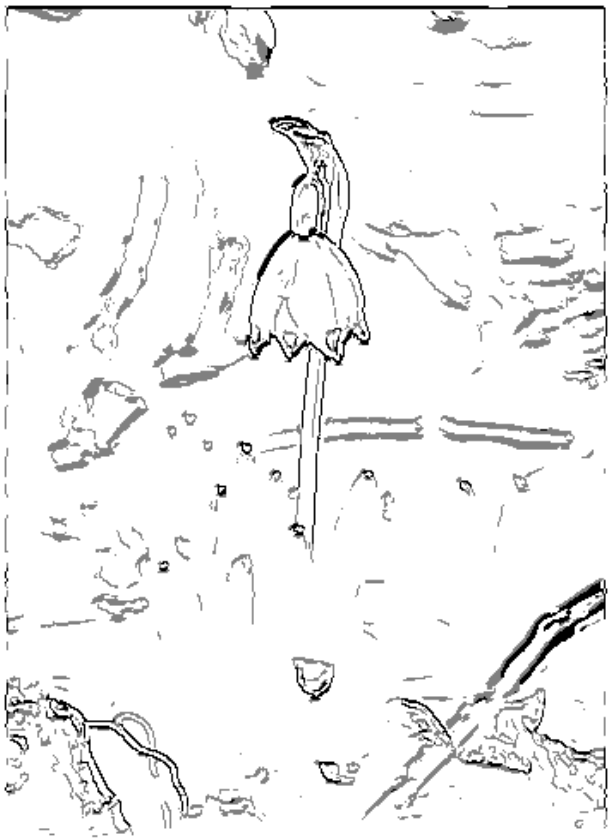


Non maximum suppression

- Gradient intensity of a pixel
  - needs to be the maximum
  - in the gradient direction
- If it‘s a maximum
  - Then apply thresholds
  - for weak & strong points



Non maximum suppression



Tracing edges

- Just retain those weak points that are connected to strong points

PHOG

- Create a histogram of *n* bins for gradient directions of edge pixels.
  - Fuzzy histogram with *n* in ~ [20, 100]
  - Good results with 40 bins
- Do not blur for Canny Edge Detector
  - Increases precision

PHOG

- Do the same for subimages
  - Four subimages per image per level
- Descriptor is rather large
  - Level 0: n bins
  - Level 1: n+4n bins
  - Level 2: n+4n+4*4n bins
  - …

Joint Histograms

- Color & texture properties per pixel
  - E.g. color (8 bins) + gradient (4 bins)
- E.g. directionality histogram per color bin
  - I.e. 4 times a color histogram

Joint Histogram

88 red pixels with a gradient direction of 0-45°

Color Histogram

	0°-45°	45°-90°	90°-135°	135°-180°	Σ
red	88	51	93	10	241
orange	29	29	53	34	144
yellow	57	58	22	88	224
green	6	13	58	8	85
blue	25	86	43	47	200
Σ	204	236	268	186	

Directionality Histogram

Joint Histograms

- Strategies
  - Find good texture properties
  - Minimize overall number of bins
- Benefits
  - Works better than pure color histograms
- Disadvantages
  - Typically slower extraction
  - Higher number of bins

Comparison of Features

- Based on the WANG SIMPLICity data set

Desc	MAP	P@10	P@20
SC(G)	0.5222	0.7692	0.7009
JCD(G)	0.5140	0.7498	0.6896
AutoColCorrel(G)	0.5099	0.7765	0.7122
FCTH(G)	0.5085	0.7390	0.6738
CEDD(G)	0.5040	0.7410	0.6794
OppHist(G)	0.4614	0.6755	0.6076
CL(G)	0.4506	0.6574	0.5903
LBP(G)	0.3699	0.6356	0.5561
RILBP(G)	0.3502	0.5748	0.5058
EH(G)	0.3454	0.5538	0.4877

Linear Search

- Corpus contains *K* images,
- Represented by feature vectors
  - f_k with *k* in [0,*K*-1]
- We have a distance funktion
  - $d(f_i, f_j)$ -> *[0, inf]*

Linear Search

- Given a query q
  - also being a feature vector
- Find a
  - ranked list of
  - the $n < K$ most relevant images
  - having minimum distance to the query

Linear Search - Approach

- Create an “index”
  - a list of “feature vectors -> image file” entries
- With a query
  - traverse list
  - (re-) order result set with fixed length

Linear Search - Complexity

- Given
  - an index of *K* image feature vectors
  - searching for *N* relevant images
- Complexity is based on
  - Taking a look at each of the *K* images
  - modifying a sorted list of *N* relevant images
- If N is “small” then
  - complexity is $O(K)$

Linear Search: How it is done in Lire

- Lire uses Lucene for storage on HDD
  - Lucene Document object
  - Lucene Field for image file path and features
- DocumentBuilder class
  - creates Document from BufferedImage
  - returns Document or Field[]
  - encapsulates feature extraction
  - stores file path in text field
  - stores feature in byte[] payload

Linear Search: How it is done in Lire

- ImageSearcher
  - encapsulates feature class & extraction
  - opens each Lucene Document
  - parses byte[] payload (done in LireFeature)
  - compares feature to query
  - manages result list
  - returns results

Linear Search: How it is done in Lire

- ImageSearcher code (Lire):

```
int docs = reader.numDocs();
for (int i = 0; i < docs; i++) {
  // ignore if deleted
  if (reader.hasDeletions() && !liveDocs.get(i))
  continue;
  Document d = reader.document(i);
  float distance = getDistance(d, lireFeature);
  assert (distance >= 0);
  // ... adapting result list
}
```

Linear Search: byte[] Representation of CEDD

```
public byte[] getByteArrayRepresentation() {
	// find out the position of the beginning of the trailing zeros.
	int position = -1;
	for (int i = 0; i < data.length; i++) {
		if (position == -1) {
			if (data[i] == 0) position = i;
		}
		else if (position > -1) {
			if (data[i]!=0) position = -1;
		}
	}
	// find out the actual length. two values in one byte, so we have to round up.
	int length = (position + 1)/2;
	if ((position+1)%2==1) length = position/2+1;
	byte[] result = new byte[length];
	for (int i = 0; i < result.length; i++) {
		tmp = ((int) (data[(i << 1)] * 2)) << 4;
		tmp = (tmp | ((int) (data[(i << 1) + 1] * 2)));
		result[i] = (byte) (tmp-128);
	}
	return result;
}
```

Linear Search:
byte[] Representation of CEDD

- CEDD has 144 bins
- Representation needs 1 byte per 2 bins
- Gets smaller for trailing zeros

- 144/2 = 72 byte per image
- ~ 68,7 MB for 1M images

Linear Search: Performance

- No **new** variables!
- Use doubles, shift operations & byte buffers.

```
public void setByteArrayRepresentation(byte[] in, int offset, int length) {
    if (in.length * 2 < data.length) Arrays.fill(data, in.length*2,
    data.length-1, 0);
    for (int i = offset; i < length; i++) {
          tmp = in[i]+128;
          data[(i << 1) +1] = ((double) (tmp & 0x000F))/2d;
          data[i << 1] = ((double) (tmp >> 4))/2d;
    }
}
```

Linear Search: How it is done in Lire

- Lucene
  - provides really fast file I/O
  - memory caching for arbitrarily large indexes
  - index management & additional text search
- Lire
  - generic use of different feature vectors
  - encapsulates (some) Lucene functions
- Runtime
  - 121,379 images indexed with CEDD
  - took 0.33 seconds search time

Note ...

- Lire supports more than that ..
  - in-memory linear search
  - parallel query processing
  - hashing

Thank you ...

... for your attention