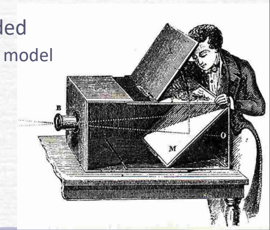
Images

- ➢Camera models
- ➢Digital images
- ➢Colour images
- ➢Noise
- ➢Smoothing

1

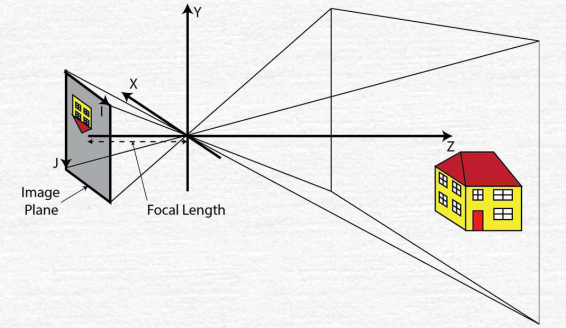
Camera models

- Components:
  - A photosensitive image plane
  - A housing
  - A lenses
- Mathematical model needed
  - The simple pinhole camera model
  - Distortions



2

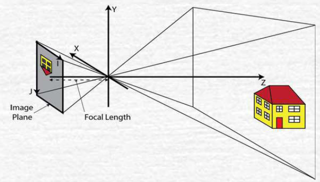
Camera models – Simple Pinhole Model



3

Camera models – Simple Pinhole Model

$$\begin{bmatrix} i.w \\ j.w \\ w \end{bmatrix} = \begin{bmatrix} f_i & 0 & c_i \\ 0 & f_j & c_j \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$



- 3-D point
- 2-D image point
- Scaling factor
- Combination of focal length and image coordinate system
- Location of the optical centre

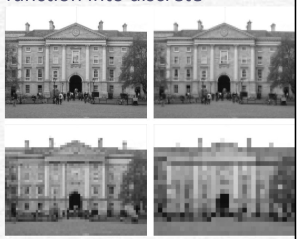
4

Digital Images

- Theoretically images are continuous 2D functions of reflected scene brightness.
  - (i, j) or (column, row) or (x, y)
- To process on a computer we need a discrete representation
  - Sample
  - Quantise

Digital Images – Sampling

- Sample the continuous 2D function into discrete elements.
- Sensor
  - 2D array
  - Photosensitive elements
  - Non photosensitive gaps
- Issues
  - Elements have a fixed area
  - Gaps



Digital Images – Sampling

- How many samples do we need ?
  - Wasted space and computation time
  - Enough for the objects of interest

```
Mat image, smaller_image;
resize( image, smaller_image,
        Size( image1.cols/2, image.rows/2 ));
```


Digital Images – Quantisation

- Represent the individual image points as digital values.
  - Typically 8 bits

Digital Images – Quantisation

- How many bits do we need?
  - Wasted space ?
  - Losing the ability to distinguish objects


```
void ChangeQuantisationGrey( Mat &image, int num_bits )
{
  CV_Assert( (image.type() == CV_8UC1) && (num_bits >= 1) &&
                                                      (num_bits <= 8) );
  uchar mask = 0xFF << (8-num_bits);
  for (int row=0; row < image.rows; row++)
    for (int col=0; col < image.cols; col++)
      image.at<uchar>(row,col) = image.at<uchar>(row,col) & mask;
}
```


Colour Images

- Luminance only
  - Simple representation
  - Humans can understand
- Colour images (luminance + chrominance)
  - Multiple channels (typically 3)
  - Around 16.8 million colours
  - More complex to process
  - Facilitate certain operations

Colour Images – Processing

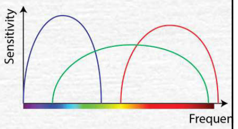
```
void InvertColour( Mat& input_image, Mat& output_image )
{
  CV_Assert( input_image.type() == CV_8UC3 );`
  output_image = input_image.clone();
  for (int row=0; row < input_image.rows; row++)
    for (int col=0; col < input_image.cols; col++)
      for (int channel=0; channel <
                input_image.channels(); channel++)
          output_image.at<Vec3b>(row,col)[channel] = 255 –
              input_image.at<Vec3b>(row,col)[channel];
}
```

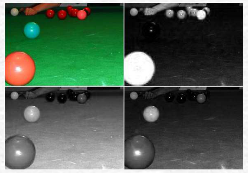

Colour Images – Efficient processing

```
int image_rows = image.rows;
int image_columns = image.cols;
for (int row=0; row < image_rows; row++) {
  uchar* value = image.ptr<uchar>(row);
  uchar* result_value = result_image.ptr<uchar>(row);
  for (int column=0; column < image_columns; column++)
  {
    *result_value++ = *value++ ^ 0xFF;
    *result_value++ = *value++ ^ 0xFF;
    *result_value++ = *value++ ^ 0xFF;
  }
}
```

Colour Images – RGB Images

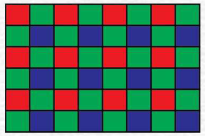
- Red-Green-Blue images
  - Most common
  - Channels correspond roughly to
    - Red (700nm)
    - Green (546nm)
    - Blue (436nm)
  - Colours combined in display





Colour Images – RGB Images

- Converting to Greyscale
  - $Y = 0.299R + 0.587G + 0.114B$
- Camera photosensitive elements
  - Separate Red, Green & Blue elements
  - Sometimes sensitive to all visible wavelengths
  - Bayer pattern:

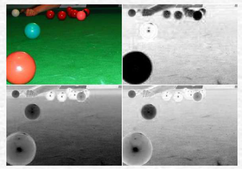


Colour Images – RGB Images

```
Mat bgr_image, grey_image;
cvtColor(bgr_image, grey_image, CV_BGR2GRAY);
vector<Mat> bgr_images(3);
split(bgr_image, bgr_images);
Mat& blue_image = bgr_images[0];
```


Colour Images – CMY Images

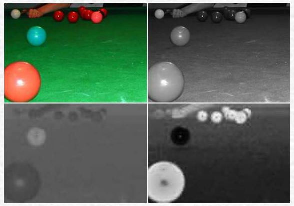
- Cyan-Magenta-Yellow images
  - Secondary colours
  - Subtractive colour scheme
    - $C = 255 – R$
    - $M = 255 – G$
    - $Y = 255 – B$
  - Often used in printers



CMY is not directly supported in OpenCV.

Colour Images – YUV Images

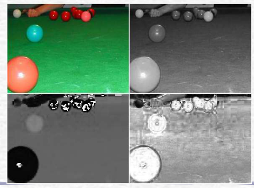
- Used for analogue television signals
  - PAL, NTSC
  - 4 Y to 1 U to 1 V
- Conversion from RGB
  - Y = 0.299R + 0.587G + 0.114B
  - U = 0.492 * (B-Y)
  - V = 0.877 * (R-Y)

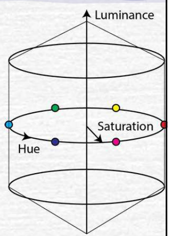


17

Colour Images – HLS Images

- Hue-Luminance-Saturation images
  - Separates Luminance & Chrominance
  - Values we humans can relate to…
    - Hue 0°..360°
    - Luminance 0..1
    - Saturation 0..1
  - Watch out for circular Hue…





18

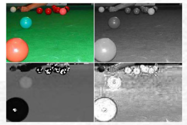
Colour Images – HLS Images

- Conversion from RGB

$$L = {Max(R,G,B) + Min(R,G,B)}/_{2}$$

$$S = \begin{cases} {Max(R,G,B) - Min(R,G,B)}/_{Max(R,G,B) + Min(R,G,B)} & \text{if } L < 0.5 \\ {Max(R,G,B) - Min(R,G,B)}/_{2 - (Max(R,G,B) + Min(R,G,B))} & \text{if } L \geq 0.5 \end{cases}$$

$$H = \begin{cases} {60.(G-B)}/_{S} & \text{if R} = Max(R,G,B) \\ 120 + {60.(B-R)}/_{S} & \text{if G} = Max(R,G,B) \\ 240 + {60.(R-G)}/_{S} & \text{if B} = Max(R,G,B) \end{cases}$$



```
cvtColor(bgr_image, hls_image, CV_BGR2HLS);
// Hue ranges from 0 to 179.
```


19

Colour Images – Other colour spaces

- HSV
- YCrCb
- CIE XYZ
- CIE L*u*v*
- CIE L*a*b*
- Bayer

20

Noise

- Affects most images
- Degrades the image
- Can cause problems with processing
- Causes?
- Measuring noise: $S/N\ ratio = \sum_{(i,j)} f^2(i,j) \Big/ \sum_{(i,j)} v^2(i,j)$
- Correcting noise…
- Types
  - Gaussian
  - Salt and Pepper

21

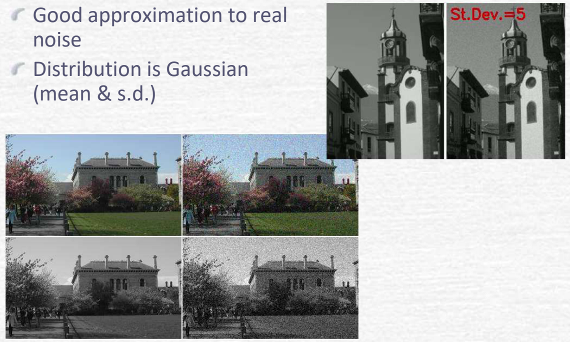
Noise – Salt and Pepper Noise

- Impulse noise
- Noise is maximum or minimum values

22

Noise – Gaussian Noise

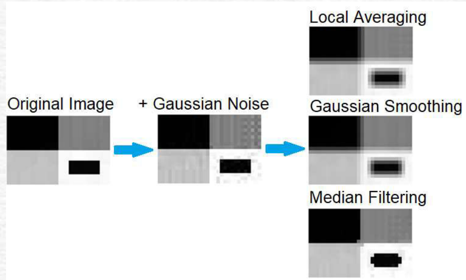
- Good approximation to real noise
- Distribution is Gaussian (mean & s.d.)



23

Smoothing

- Removing or reducing noise…
- Linear & non-linear transformations



24

Smoothing – Averaging Filters (linear)

- Linear transformation (convolution)
- Local neighbourhood
  - $f(i,j) = \sum\sum_{(m,n)\in O} h(i-m, j-n).g(m,n)$
  - Different masks…
    - Local Average
    - Gaussian

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$h = \frac{1}{10}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

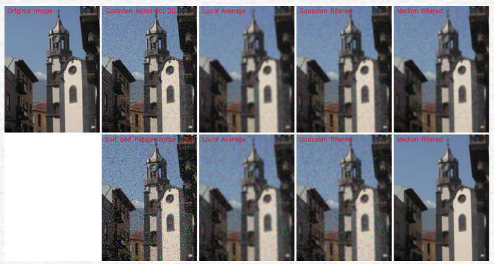
$$h = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

- Acceptable results?
  - Suppression of (small) image noise
  - Blurring of edges

```
blur(image,smoothed_image,Size(3,3));
GaussianBlur(image,smoothed_image,Size(5,5),1.5);
```


25

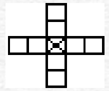
Smoothing examples



26

Smoothing – Median Filter (non-linear)

- Use the median value… **11 18 20 21 23 25 25 30 250**
  Median = 23 Average = 47
- Not affected by noise
- Doesn't blur edges much
- Can be applied iteratively
- Damages thin lines and sharp corners
  - Change region shape



- Computational expensive
  - Standard – $O(r^2 \log r)$
  - Huang – $O(r)$
  - Perreault (2007) – $O(1)$

```
medianBlur(image, smoothed_image, 5);
```


27

Smoothing – Effects of mask size

28

Image Pyramids

- To process images
  - At multiple scales
  - Efficiently
- Technique
  - Smooth image (often Gaussian)
  - Subsample (usually by a factor of 2)

```
pyrDown( image, smaller_image,
  Size( (image1.cols+1)/2, (image.rows+1)/2 ));
```