

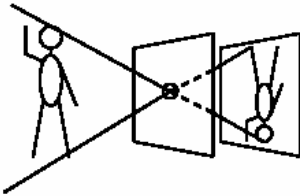


Revision Notes

Images and cameras

Cameras

A Simple Camera Model



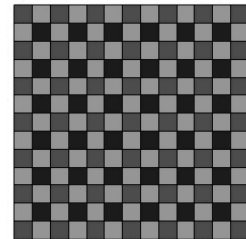
The 'pinhole' camera – light passes through a small hole and falls on an image plane. The image is inverted and scaled.

Real cameras do not have 'pinholes' as the size of the hole means very little light is admitted resulting in long exposure times. However, larger holes lead to blurred images. The solution – have a larger hole but with a lens to make it act like a small hole.

Forming the Image

Traditional Cameras use film which reacts chemically with light. Digital cameras use a 2D array of CCD (charge coupled devices) which emit electrons when hit with light. Each CCD element can have a filter to make it sensitive to a particular colour (red, green or blue).

This filtering is commonly done by having the CCD array covered in a Bayer pattern (right). There are as many green pixels as red and blue added together – this is because the human eye is more sensitive to green light.



To read a CCD array, each row is moved in turn into a serial register. Each pixel is read off from the register.

Image representation in a computer

A digital image is a 2D array of pixels – each pixel may either have a single (grey) value, or several colour values (typically Red, Green and Blue). Each colour channel of each pixel holds one byte of information (0 – 255 is represented by 8 bits = 1 byte).

File formats and compression

There are two basic methods for a computer to render, or store and display, an image. When you save an image in a specific format you are creating either a raster or meta/vector graphic format. We shall be considering the Raster image format – that which breaks the image into a series of pixels.

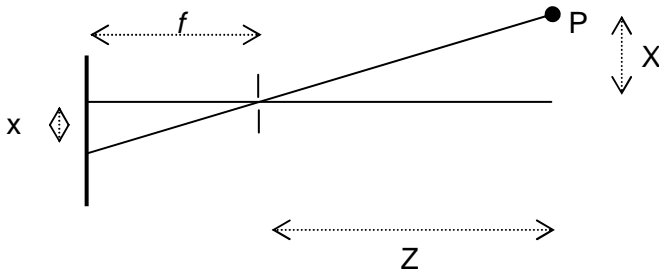
Compression

The GIF (Graphics Interchange Format) compression algorithm uses a 256 colour depth – the 256 colours that best fit the image are chosen. Each pixel is assigned a colour. This reduces the size to one third original. The image is then put through a lossless compression algorithm (although we have already lost some information) called LZW.

The JPEG (Joint Photographic Experts Group) format is based on studies of how people actually perceive images. Some information can be discarded without changing the appearance much. It is a lossy compression, but the human does not perceive much difference. Images are encoded into 8x8 blocks. A discrete cosine transform is applied to separate the image into parts (or spectral sub-bands) of differing importance (with respect to the image's visual quality). The results are quantised to remove unused frequencies. The result is encoded using a variable length encoding scheme (encoding high probability values with shorter bit streams).

Stereo

The Perspective Camera



By Trigonometry (congruent triangles):

$$x = f \cdot (X/Z)$$

$$y = f \cdot (Y/Z)$$

Binocular Stereo under Parallel Camera Geometry

Such a system is given by two cameras where:

- i) the axes are parallel
- ii) focal lengths are equal
- iii) image plains are coplanar.

This allows us to work out the 3D coordinates of a point in space from the coordinates of this point on the image given the two cameras.

x and y can be worked out as above.
z is given by the formula $2hf / (P_r - P_l)$

The Four Stages of Binocular Stereo

1. Choose/constrain camera geometry
2. Select matching primitives – choose image features of real world points.
3. Correspondence – match points in left image to those in right image
4. Interpretation

Noise, filters, and experiments

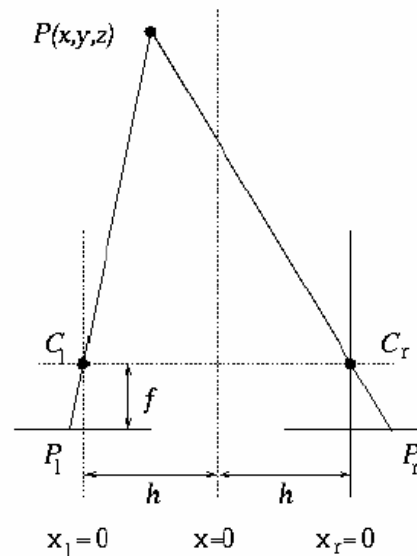
Noise is a result of several factors: i) Digital images are quantized (left), ii) JPEG uses lossy compression, iii) imperfect sensors.

Types of noise

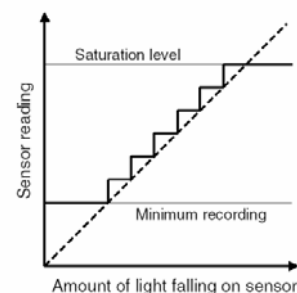
Salt and Pepper Noise: some pixels are incorrectly white (false saturation), some are incorrectly black (failed response).

Uniform Noise: Pixel values are usually close to their true value. The average value is equal to the true value. We get a range of values around this point – each value is equally likely.

Gaussian Noise: Pixel values often a little off the true value. On average, they give the right value. They give values closer to the true value rather than far away.



by



(see



Noise removal

We remove noise by filters. Many involve considering a 'neighbourhood' of pixels and changing the value of the center pixel.

The *mean* filter takes the mean of all pixels in the neighbourhood and sets the center pixel to this value. Especially effective with Uniform noise.

The *median* filter takes the median value of all pixels in the neighbourhood and sets the center pixel to this value. Especially effective with Salt and Pepper noise.

The *Gaussian* filter produces a grid of numbers for each in the neighbourhood. Each pixel is multiplied by it's corresponding number and the sum placed in the middle.

The formula for the Gaussian numbers is
$$P(x,y) = \frac{1}{\sigma^2 2\pi} e^{-\frac{(x^2+y^2)}{2\sigma^2}}$$

σ^2 = variance

The Gaussian curve extends infinitely in all directions. The volume under the curve is 1. As we are only considering a finite neighbourhood we must normalize the result so that the sum of the grid numbers is still 1 – as this conserves the overall intensity of the pixel.

Anisotropic Diffusion

Mean and Gaussian filters make each pixel more like it's neighbours. Anisotropic diffusion makes each pixel more the those neighbours which it is already similar to.

We use a similarity function, S(p,q), which has values into the range 0-1 (the closer to 1, the more similar p and q). This function is used to compute a weighted average pixel value.

Edges

We encounter problems when using a neighbourhood window on edge pixels. Several solutions:

- i) Ignore them
- ii) Apply different filters to the edges
- iii) Extend the image for processing (linear extrapolation outwards, or 'nearest neighbour')

Experimental design

Hypothesis → Testing → Conclusions

Factors to consider in an Image/Vision Experiment:

- External parameters: lighting, cameras, etc.
- Algorithm parameters: thresholds, etc., colour space
- Images used to build models
- Images used to test algorithm – good range.

Errors – false positive and false negative

Colour models and thresholding

Different colour representations

RGB (Red Green Blue)

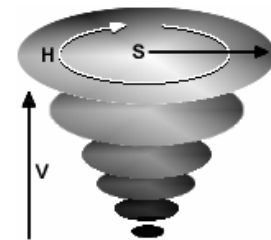
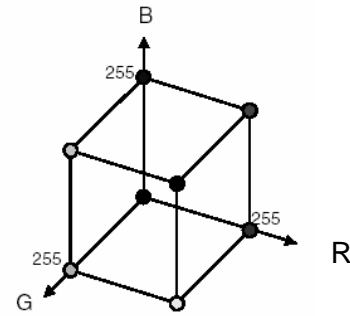
- linked to the way our eyes work
- used with computer monitors, etc.
- colours of *light* not of pigment

Greyscale – single value for each pixel (same value for all three channels). However, our eyes are more sensitive to green light, so a weighted formula can be used:

$$i = 0.30r + 0.59g + 0.11b$$

HSV (Hue Saturation Value)

- based on colour, rather than light.
- often used for colour analysis
- visualised as a cone with black at the tip, and a colour wheel at the base with white in the centre and stronger colour around the outside. Greyscale down the centre.
- Hue: what general colour it is
- Saturation: how strongly coloured
- Value: how bright/dark



Converting from RGB to HSV:

Assuming: RGB values in the range 0 – 1
H values in the range 0 – 360 (degrees)
S and V in the range 0 – 1.

$$V = \text{Max}(R, G, B)$$

$$S = (\text{Max}(R,G,B) - \text{Min}(R,G,B)) / (\text{Max}(R,G,B))$$

If $\text{Max}(R,G,B) = R$, we know we're between magenta and yellow.
Hence $H = (60 \times (G - B)) / (\text{Max}(R,G,B) - \text{Min}(R,G,B))$

If $\text{Max}(R,G,B) = G$
 $H = 120 + (60 \times (G - B)) / (\text{Max}(R,G,B) - \text{Min}(R,G,B))$

If $\text{Max}(R,G,B) = B$
 $H = 240 + (60 \times (G - B)) / (\text{Max}(R,G,B) - \text{Min}(R,G,B))$

Converting from HSV to RGB

If $S = 0, R = G = B = V$

Else

The general colour, C, is given by $H/60$

The fractional part lost, $F = H - 60.C$

Now calculate:

$$P = V \times (1 - S)$$

$$Q = V \times (1 - S \times F)$$

$$T = V \times (1 - S \times (1 - F))$$

Then use the table, right.

C	R	G	B
0	V	T	P
1	Q	V	P
2	P	V	T
3	P	Q	V
4	T	P	V
5	V	P	Q

CMYK (Cyan Magenta Yellow 'Black')

- Used for printing
- 'Subtractive colours': Cyan absorbs red light, Magenta absorbs green light, yellow absorbs blue light.

$$M + Y = R$$

$$C + Y = G$$

$$M + Y = B$$

Theoretically, $C + M + Y = K$, but usually a true black ink is used.



Revision Notes

- Assuming RGB and CMYK are in the range 0 – 1:
 - R = 1 – C
 - G = 1 - M
 - B = 1 – Y

CIE (Commission Internationale de l'Éclairage)

- Sets standards for pigments and displays
- Based on physical properties of light
- Three components: intensity/luminance and two colour coordinates.

YUV

- Used for (PAL) television signals
- Y is intensity; U and V are the colour channels. U is blue-yellow axis. V is red-green axis.

In a TV signal, more emphasis is given to Y – human vision is more sensitive to intensity than colour.

YUV to/from RGB

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.30 & 0.59 & 0.11 \\ -0.17 & -0.33 & 0.50 \\ 0.50 & -0.42 & -0.08 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Thresholding images and processed images

A threshold is often found by drawing a histogram and choosing a threshold value between two peaks. Textures are most easily classified by their standard deviations.

Finding Connected Components

Inputs: P and I (where P is a pixel on the image and I is the segmented image)

Make two lists, object and search.

Add P to search

While (search is not empty) {

 Remove a pixel, q, from search

 Add q to object

 For each neighbour, n, of q {

 If n is not in object AND I(n) = I(p) {

 Add it to search

 }

 }

}

Region Growing

- Compute statistics (colour, standard deviation, etc.) about a seed region
- Check neighbours to see if they can be added
- Recompute statistics

Split and Merge

- A region that is not uniform is split into separate parts.
- Two adjacent regions that are similar are merged together.
- Repeat until no regions need be split or merged.

Indexing and retrieval

Two methods: i) Text-based – each image annotated with relevant key words: subjective and laborious.
ii) Content-based image indexing and retrieval (CBIR) – identifies using visual features (colour, texture, shape, etc.).

The difference between the high level concept of what an image portrays and the numerical, low level, representation of the image is called the *semantic gap*.

Colour is a surprisingly effective visual cue.

Image histograms

Colour Histogram Construction

- Colour quantization: Divide each colour axis into equal length sections. Map each colour into it's corresponding bin - Bin(red,green,blue)
Images are 24-bit – 3 colour channels each with 256 possible values – $256^3 = 2^{24}$
A 512 bin histogram would accordingly divide each channel into $512^{1/3}$ parts.
An 8 bin histogram would divide each channel into 2. ($2 \times 2 \times 2 = 8$)
- Plot histogram (Bins across the x, frequency on the y)

Similar images have similar histograms.

Histogram difference computations

Histogram Intersection: $(H_1, H_2) = \sum_i \min(H_1, H_2)_i$ – sum of minimum values

Euclidean/Straight-line distance: $\sqrt{\sum_i (H_1 - H_2)_i^2}$ – root of sum of the squared differences

Manhattan Distance: $\sum_i |(H_1 - H_2)_i|$ - sum of absolute differences

Problems

- If all colours in one image are slightly different from query image, their histograms may have nothing in common.
- Colour of light
- Doesn't take into account colour distribution

Other methods

Region/Object Based Query

- Divide target into windows. Select windows with similar colour distribution to query.

Back-projection

Highlight all colours in the image that are similar to the colours being sought.

Practical image processing

Programming with Images

Images are often intended for humans to view. The aim is often to reduce file size without changing the appearance. For vision or image processing we want raw data. In a program, we might typically want to read and write image files, create new images, determine the image's size and query/set certain pixel values. This class provides basic image manipulation routines. This class is essentially a wrapper around a BufferedImage, and provides simple methods for reading and writing images in JPEG format, accessing pixel values, and creating new images.

Horizontal Flip

Read imageIn from file

imageOut = new image the same size as imageIn

```
for x = 0 to imageIn.width {  
  for y = 0 to imageIn.height {  
    imageOut(width-x, y) = imageIn(x, y)  
  }  
}
```

Write imageOut to file

or

Read image from file

```
for x = 0 to width/2 {  
  for y = 0 to height {  
    swap image(x,y) and image(width-x,y)  
  }  
}
```

Write image to file

Rotate (90° anti)

Read imageIn from a file

imageOut = new image imageIn.width high and imageIn.height wide

```
for x = 0 to imageIn.width {  
  for y = 0 to imageIn.height {  
    imageOut(y,x) = imageIn(imageIn.width-x,y) // n.b. flip as well  
  }  
}
```

Write imageOut to file



Revision Notes

Sharpen Filter

Uses a mask to accentuate the difference between the center pixel and the surrounding pixels.
Sets the center pixel to it's present value multiplied by a factor k minus the sum of all the surrounding pixels:

```
for (x = 1; x < width-1; x++) {
  for (y = 1; y < height-1; y++) {

    for (dx = -r; dx <= r; dx++) {
      for (dy = -r; dy <= r; dy++) {

        if (dy==0 && dx==0) continue ; // skip the center pixel
        sum = Image(x+dx,y+dy)*k + sum); // sum up surrounding pixels * k
      }
    }
  }
}
```

- N.B.:
- Problem of the edges
 - Only cast to integer values at the end (before setRGB) to avoid rounding errors
 - Need to check that values are between 0 –255

Enlarging an Image

Simple algorithm:

Read imageIn from file
imageOut = new image twice the size as imageIn

```
for x = 0 to imageOut.width {
  for y = 0 to imageOut.height {
    imageOut(x, y) = imageIn(x/2, y/2)
  }
}
```

Write imageOut to file

However, this means that each pixel in the original image becomes four in the new one leading to a blocky result. A solution is to use the average of the pixels in question.

If x and y are even, we're ok.

If x is odd and y is even, pixel (x',y') becomes the average of (x/2+1,y/2) and (x/2,y/2).

If x is even and y is odd, pixel (x',y') becomes the average of (x/2, y/2+1) and (x/2, y/2).

If x and y are odd, pixel (x',y') becomes the average of all four of the above pixels.

Amended algorithm:

Read imageIn from file
imageOut = new image twice the size as imageIn

```
for x = 0 to imageOut.width {
  for y = 0 to imageOut.height {
    if (x is even and y is even) imageOut(x, y) = imageIn(x/2, y/2)
    else {
      if (x is odd but y is even) {
        imageOut(x, y) = average(imageIn(x/2, y/2), imageIn(x/2+1, y/2))
      }
    }
  }
}
```




Revision Notes

Difference Image

This simply takes the difference between two images:

```
for (x = 0 to width) {  
    for (y = 0 to height) {  
        diff(x,y) = frame1(x,y) - frame2(x,y)  
    }  
}
```

However, the resulting image will have values in the range $-255 - 255$. To options – i) take the absolute difference, ii) Scale the results (divide by 2 and add 128)

The result can then be thresholded for more a more helpful result.

Histograms

Calculation:

```
hist[] = new int[256]  
initialise every element of hist to 0;  
for (x = 0 to width) {  
    for (y = 0 to height) {  
        i = intensity at (x,y)  
        increment hist[i]  
    }  
}
```

Now, the array will hold the frequency of each intensity. E.g., the number of times a pixel of intensity 128 was found is held in `hist[128]`.

Displaying:

hist = a histogram of the image in an array
HistImage = new WxH image
Set all pixels in HistImage to be White

maxHist = largest value in hist

```
for (x = 0 to W-1) {  
    height of bar = H *(maxHist - hist) / maxHist  
    for (y = H-1 down to height) {  
        set HistImage(x,y) to be black  
    }  
}
```

Revision Notes

We use (maxHist - hist) because we are drawing up from the bottom of the image, so need to find how much white space to leave, rather than how much black line to draw. Multiplying by H and then dividing by maxHist scales the line so that it fits in the image nicely.

Background Images

To find the background image of a sequence of frames from a fixed video camera, simply take the mean value of the pixel from all the frames:

```
for (all the frames) {  
    for (a=0 to width) {  
        for (b=0 to height) {  
            imageBackground(a,b) = imageBackground(a,b) + frame(a,b)  
        }  
    }  
}  
  
for (each pixel in imageBackground) {  
    imageBackground(a,b) = imageBackground(a,b) / number of frames  
}
```

In practice, it may be easier to read the intermediate values into an array.

Correlation

The Correlation between two images is a number representing how similar they are. We could use Euclidean distance: $D^2 = \sum (\text{ImageOne}(x,y) - \text{ImageTwo}(x,y))^2$. The similarity is then $1/D$. However, if we expand the Euclidean distance, it is proportional to simply:
 $\sum (\text{ImageOne}(x,y) * \text{ImageTwo}(x,y))$

Read in two images, image1 and image2
Check that their dimensions are the same
correlation = 0;

```
for x = 0 to image1.width{  
    for y = 0 to image1.height{  
        correlation = correlation + (image1(x, y) * image2(x, y))  
    }  
}
```

N.B. Problems can arise if one image is systematically brighter than the other. To solve this, one can subtract the mean grey level of the whole image from each pixel reading – Normalised Correlation.



Revision Notes

Template Matching

Find a template in a target image:

```
for x = 0 to target.width-template.width{
  for y = 0 to target.height-template.height{

    correlation = correlate(template, target, x, y)
    if (correlation < min_correlation) {
      min_correlation = correlation
      min_x = x
      min_y = y
    }
  }
}

for x = 0 to template.width {
  for y = 0 to template.height {
    target(min_x + x, min_y + y) = some distinctive colour
  }
}
```

