Applied Computer Vision

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 3

Image Acquisition and Representation

Image Acquisition and Representation

Any visual scene can be represented by a continuous function (in two dimensions) of some analogue quantity

- This is typically the reflectance function of the scene:
  the light reflected at each visible point in the scene

- Other functions: depth (i.e. distance from the camera) or disparity

Image Acquisition and Representation

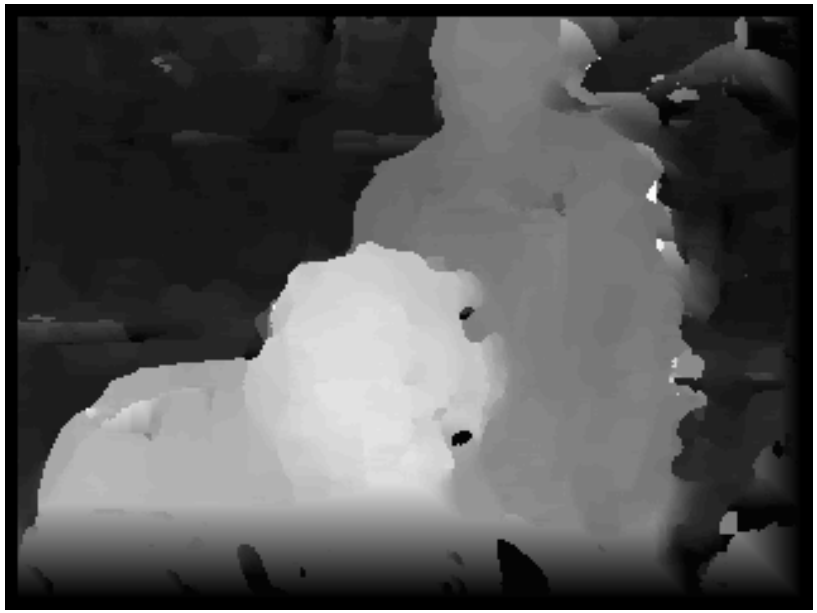
Such a representation is referred to as an image:

> the value at any point in the image corresponds to the intensity of the reflectance function at that point

Digital images represent the reflectance function of a scene but they do so in a sampled and quantised form

Each quantized integer value is known as a pixel and is the smallest discrete accessible sub-section of a digital image.

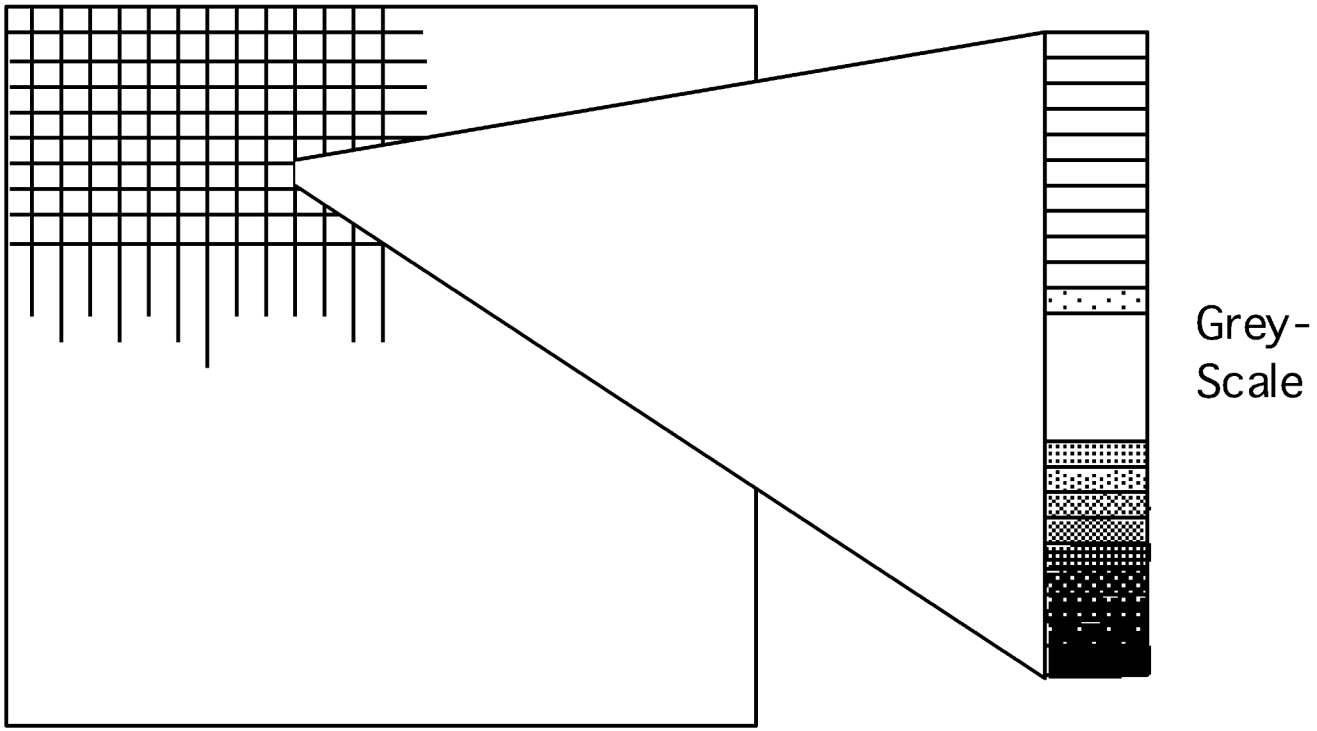
Image Acquisition and Representation



Intensity image (left) and range/depth image extracted from three intensity images [PointGrey]

Sampling and Quantization

Rectangular Sampling Pattern

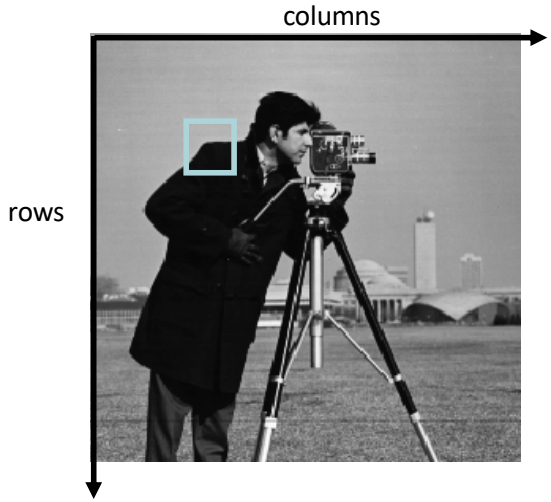


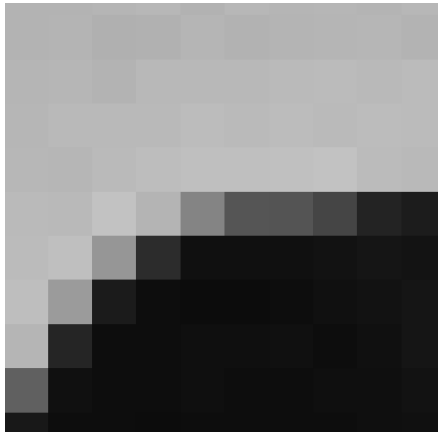
Sampling and Quantization

The number of grey-levels in the (equally-spaced) grey-scale is called the quantisation or grey-scale resolution of the system.

In all cases, the grey-scale is bounded by two grey-levels, black and white, corresponding to zero intensity and the maximum measurable intensity, respectively

Sampling and Quantization

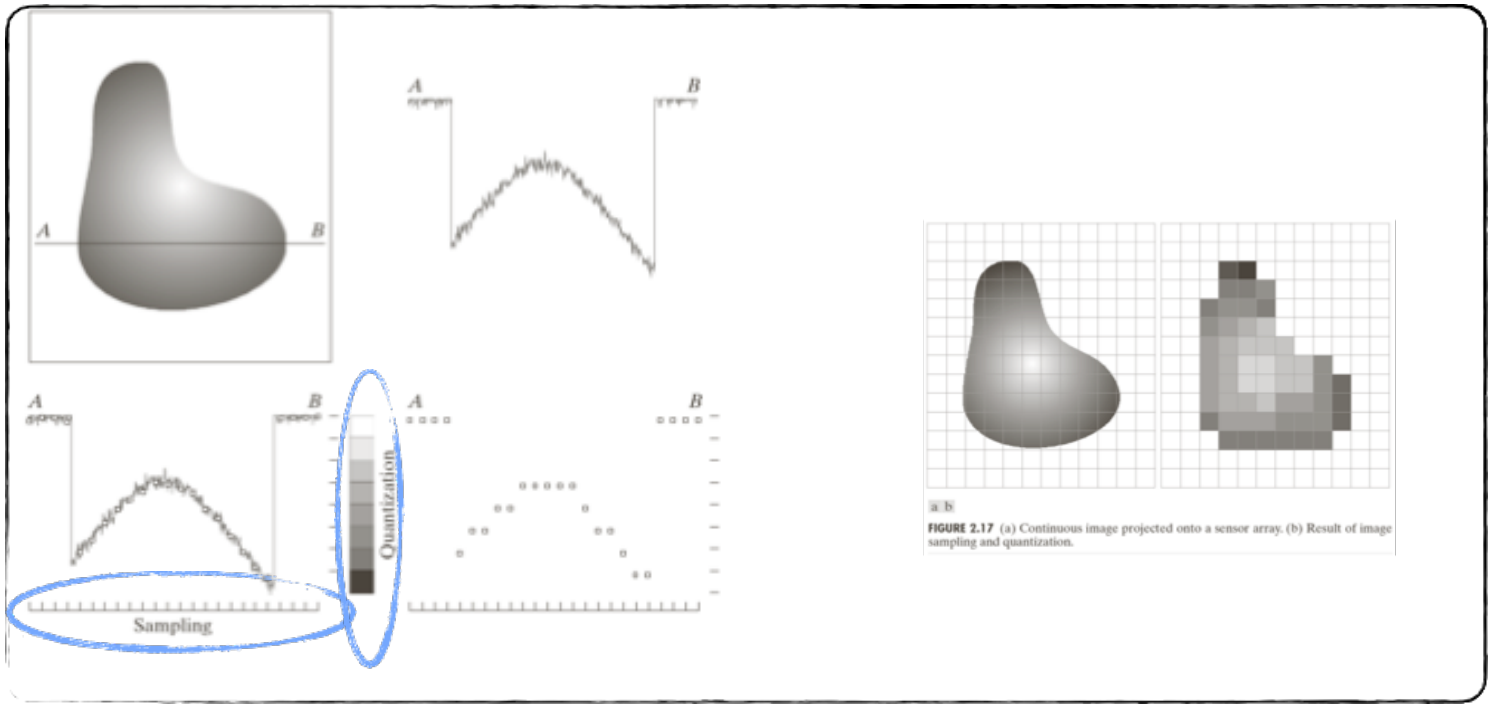




180	183	181	181	179	181	180	180	182	181	183	181	183	184	184	184	184	181	184	183	180
178	178	178	177	182	182	180	177	182	180	183	179	181	181	183	180	181	180	179	180	180
179	178	178	180	177	178	179	179	181	183	178	183	181	182	180	183	181	181	180	181	182
179	179	177	179	177	171	179	180	176	177	181	178	180	181	182	179	183	181	179	177	179
180	179	180	181	180	181	181	182	179	184	184	184	186	187	185	187	182	183	187	186	182
178	179	181	182	180	182	182	185	185	185	188	186	188	186	188	187	190	191	188	186	186
181	178	181	178	186	180	183	182	186	189	191	191	192	194	187	185	178	178	180	184	188
179	177	180	181	180	183	186	185	194	180	132	85	84	69	35	28	25	25	27	29	41
176	177	181	180	183	181	187	191	150	44	16	16	17	18	21	19	19	18	20	19	21
181	181	180	181	181	183	190	155	27	13	12	12	13	16	18	21	18	19	19	19	17
179	179	181	179	186	185	181	37	14	14	15	15	16	14	16	21	19	18	20	19	19
182	180	184	182	183	195	96	16	14	14	15	14	14	15	15	18	20	17	19	18	18
180	180	183	186	191	159	24	13	13	12	13	14	14	14	15	16	20	17	19	19	17
181	183	185	190	185	49	15	16	16	13	12	12	13	14	14	14	19	17	17	18	19
184	184	187	192	104	22	20	17	16	15	12	14	13	15	16	16	18	19	18	18	18
186	189	193	133	24	17	16	16	15	15	16	13	13	15	16	16	16	17	17	16	18
188	192	137	31	16	16	16	17	18	18	15	14	13	14	15	16	17	16	17	16	16
193	126	26	17	17	16	16	16	17	16	16	14	14	13	14	16	16	15	15	16	16
131	24	16	15	16	16	16	16	16	16	17	15	14	12	13	14	15	15	14	16	16
24	17	17	15	16	16	16	17	16	16	17	16	15	15	14	15	15	17	15	15	16
15	15	13	13	15	17	16	15	15	15	14	13	13	13	13	14	14	14	14	15	14

Credit: Francesca Odone, University of Genova

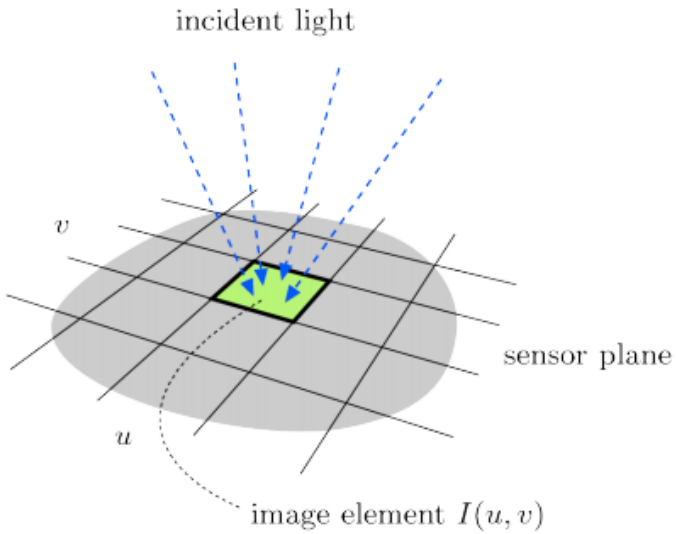
Sampling and Quantization



Credit: Francesca Odone, University of Genova

Sampling and Quantization

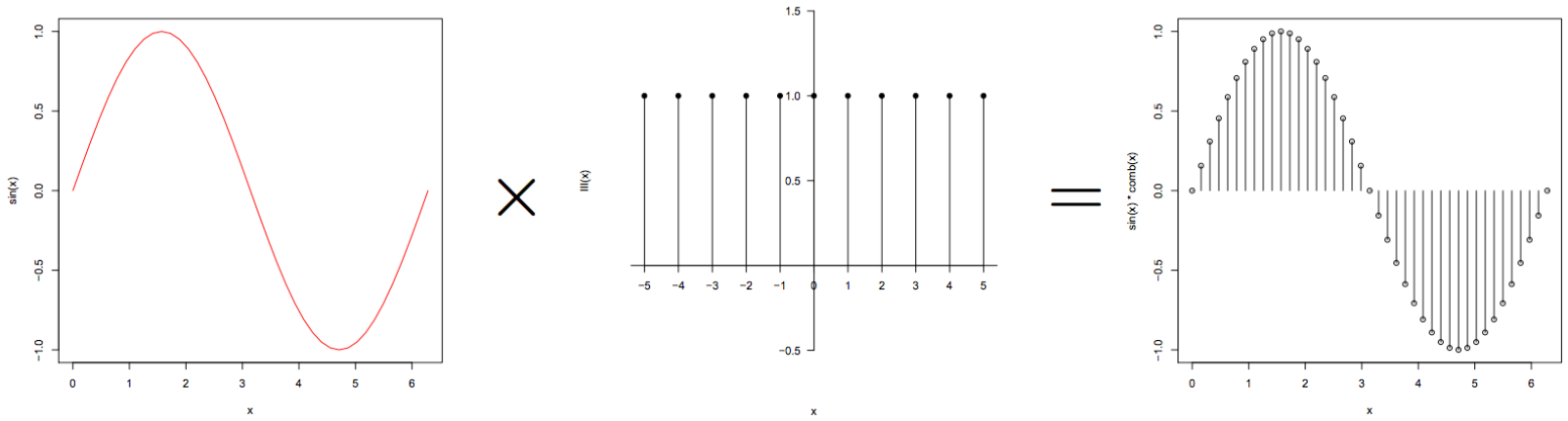
- Let us imagine an image to be sampled and quantized is overlaid with a regular grid
- This grid is referred to as sampling grid
- Each element of the grid will contain a portion (region) of the image. The whole portion will be approximated by a unique (average) value
- A coarse sampling grid produces an image with fewer details



Credit: Francesca Odone, University of Genova

Sampling and Quantization

We can think of spatial sampling as a multiplication of a continuous signal with a comb function



Credit: Francesca Odone, University of Genova

Sampling and Quantization

The sampling frequency needs to be matched to the frequency content of the scene ...

Shannon's Sampling Theorem:

the minimum sampling frequency required to reconstruct a signal from its instantaneous samples must be at least twice the highest frequency

$$f_{\mathrm{s}} \geq 2 f_{\mathrm{max}}$$

The maximum frequency in the signal is known as the Nyquist Frequency

The reciprocal of the minimum sampling frequency is known as the Nyquist Rate

$$r_{\mathrm{s}} = 1/f_{\mathrm{s}}$$

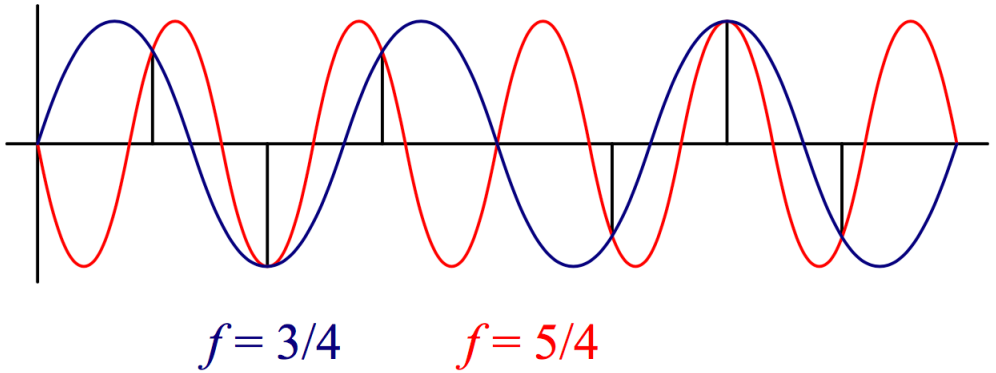
en.wikipedia.org/wiki/Aliasing

Sampling and Quantization

If you sample at less than the Nyquist rate, the sampled image will be aliased:

Different signals become indistinguishable (or *aliases* of one another)

Consider the blue sine wave of frequency $f = 3/4$ and the red of $f = 5/4$ when sampled a frequency $f = 2$



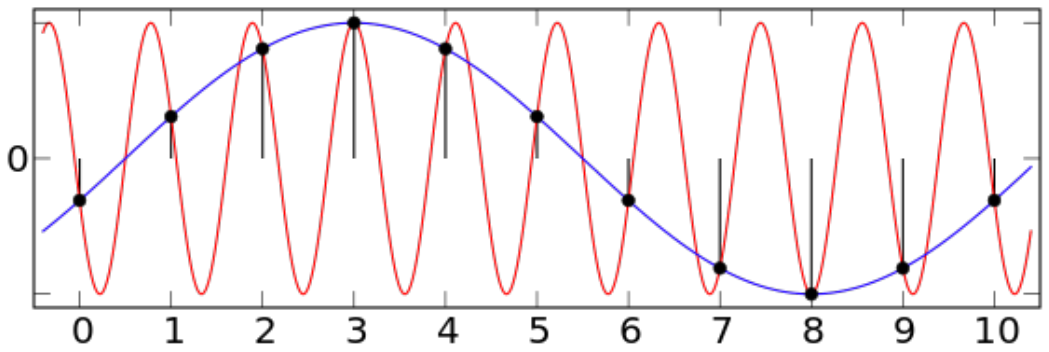
Credit: R. Szeliski, *Computer Vision: Algorithms and Applications,* Springer, 2010.

Sampling and Quantization

If you sample at less than the Nyquist rate, the sampled image will be aliased:

Aliasing: the artifact that results when the signal reconstructed from samples is different from the original continuous signal i.e. introduction of low spatial frequencies that do not exist in the original signal

(in essence, the high frequencies are aliased into visible low-frequencies)

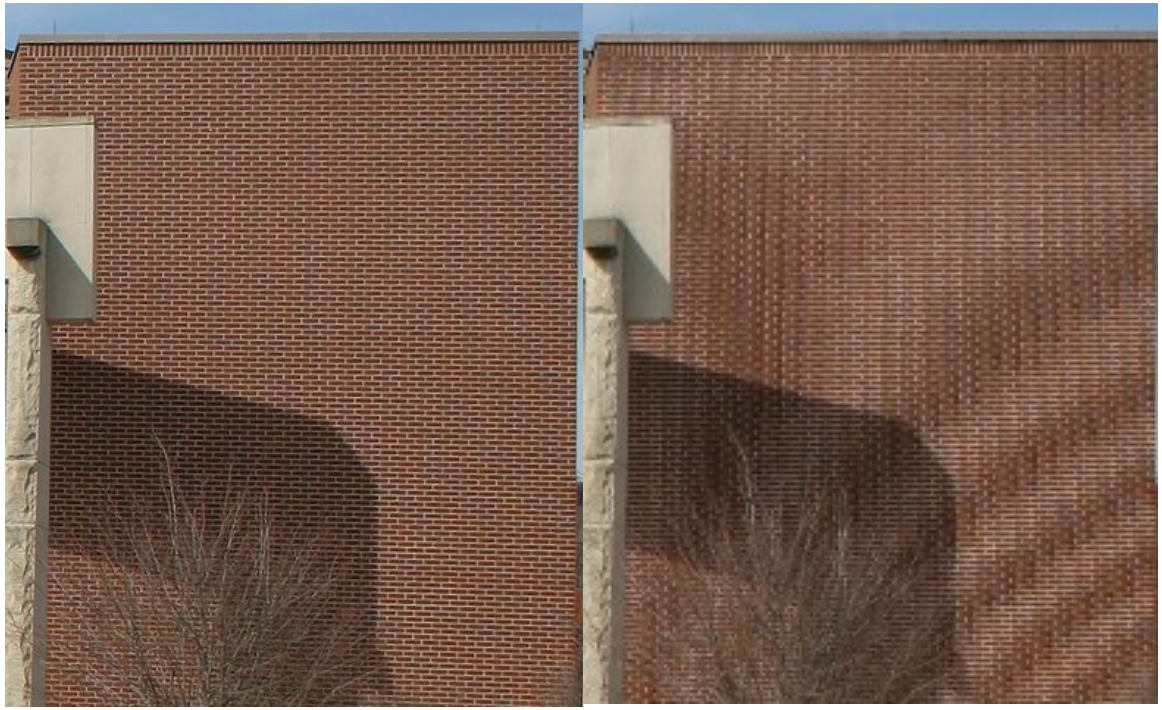


en.wikipedia.org/wiki/Aliasing

Sampling and Quantization

Aliasing: the artifact that results when the signal reconstructed from samples is different from the original continuous signal i.e. introduction of low spatial frequencies that do not exist in the original signal

Moiré pattern

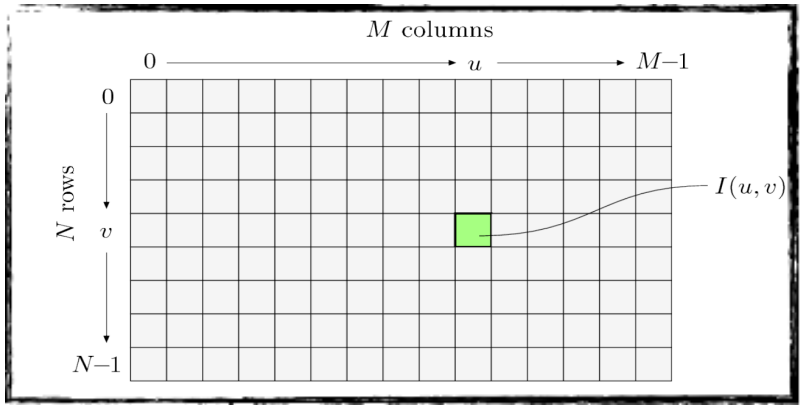


en.wikipedia.org/wiki/Aliasing

Sampling and Quantization

The size of the image is given by the number of pixels composing it

The size is conventionally expressed by the number of rows times the number of columns of the image matrix, eg 640 x 480



Credit: Francesca Odone, University of Genova

Sampling and Quantization

... and the resolution

- An image with a fixed size (in *pixels*) can be visualized at different sizes (in *mm*) on a **support** (paper, monitor, ...)
- The *visualization size* is controlled by the **resolution.**
- The resolution depends on the size of the image *and* the size of the support
- It is measured in dots/cm or, more frequently, dots/inches (dpi).

The resolution describes how *dense* are the elements on the support.

Credit: Francesca Odone, University of Genova

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

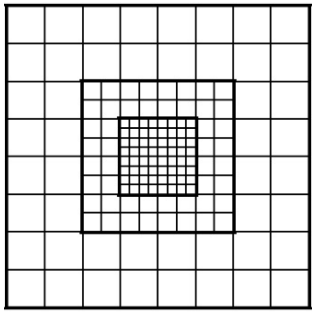
Sampling and Quantization

Image Pyramids

Sampling and Quantization

Space-variant sampling

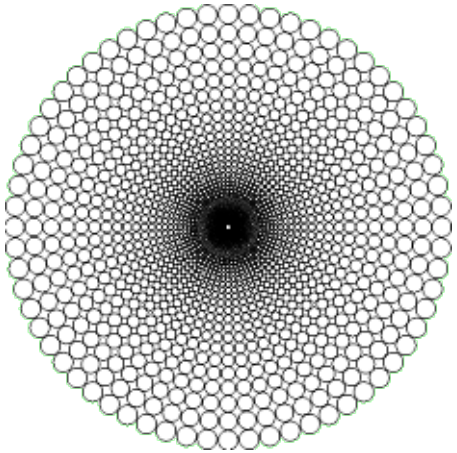
Images where resolution and field of view changes so that each image contains the same number of pixels



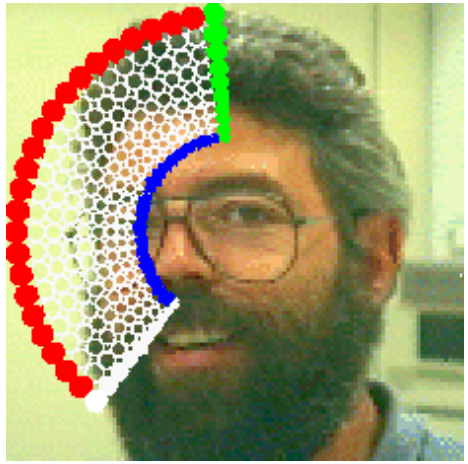
Superimposing the above images [G. Sandini]

Sampling and Quantization

Space-variant sampling: log-polar images (foveated images)



[IBIDEM retina]



Log-polar pixel tessellation [G. Sandini]

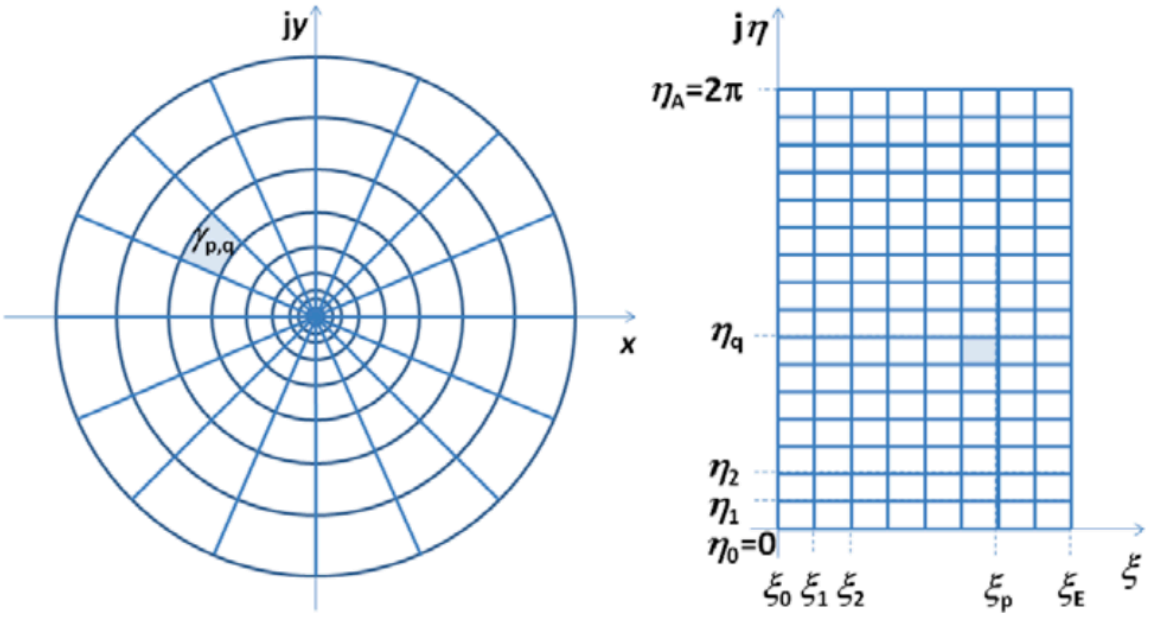
Transformation of log-polar image into a uniform, rectangular pixel tesselation [G. Sandini]

Advantage: similar to characteristics of human eye

Efficient coding, e.g. video conferencing (64 kPixel)

Sampling and Quantization

Space-variant sampling: log-polar images (foveated images)



Log-polar transform:

$$w = \log(z)$$

where

$$w = \xi + \mathrm{j}\eta$$

$$z = x + \mathrm{j}y$$

$$\xi = \log(|z|) = \log\sqrt{x^2 + y^2}$$

$$\eta = \arg(z) = \mathrm{atan2}(y, x)$$

ξ (eccentricity)

η (angle)

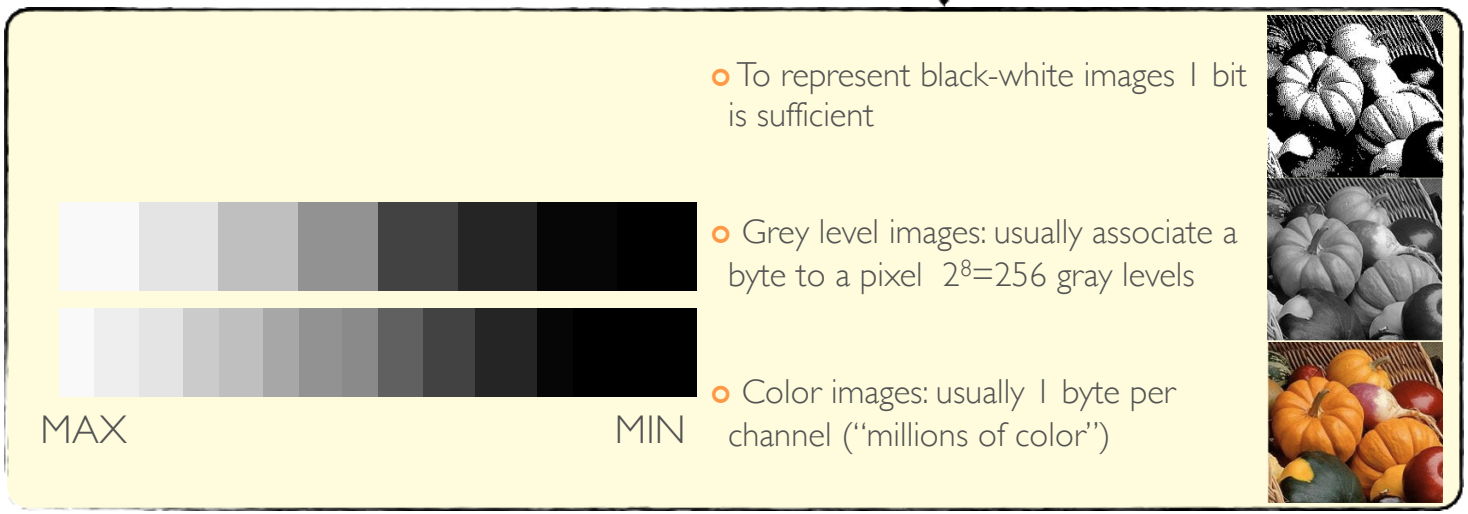
Sampling and Quantization

Dynamic range

Total number of distinctive values occurring in the image

- it is limited by the number of bit per pixel we may want to use
- it is also limited by the physical dynamic range of the sensor

this is related to the quantization process…



Credit: Francesca Odone, University of Genova

Sampling and Quantization

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Image Representation

Pixel content depends on the image type

- Gray level pictorial digital images ("black and white photos"): intensity
- Color pictorial digital images: color (modeled as triplets, eg RGB)
- Range images: depth information
- Medical images: radiation absorbance level
- Thermal images: heat

Credit: Francesca Odone, University of Genova

Image Representation

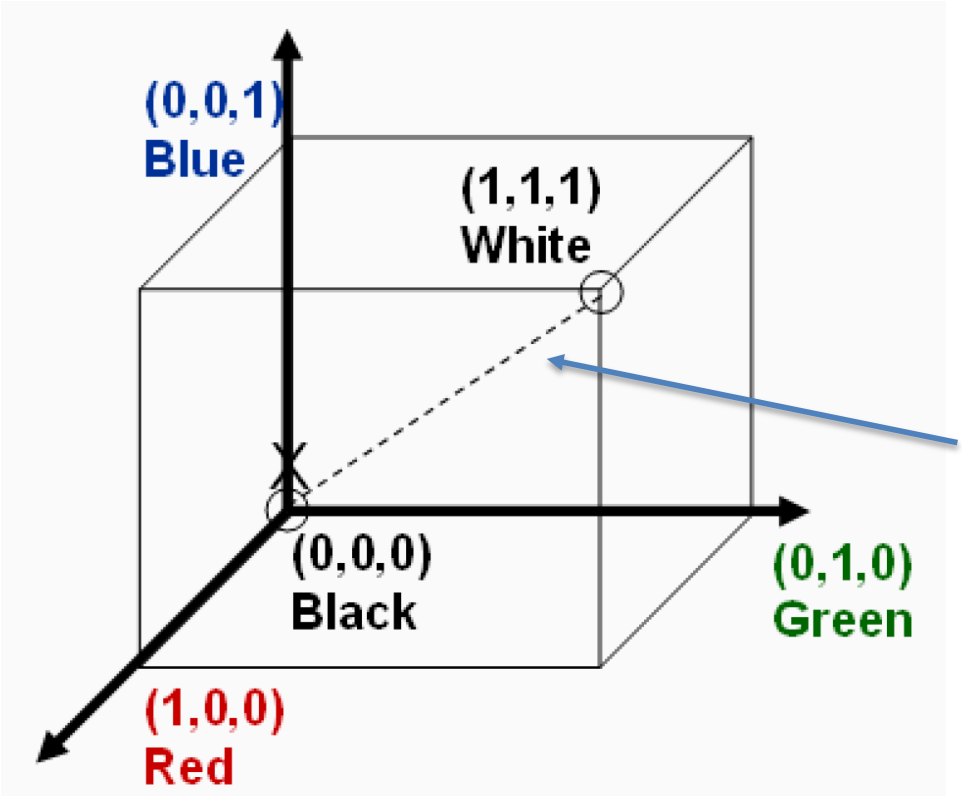
Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Colour Spaces

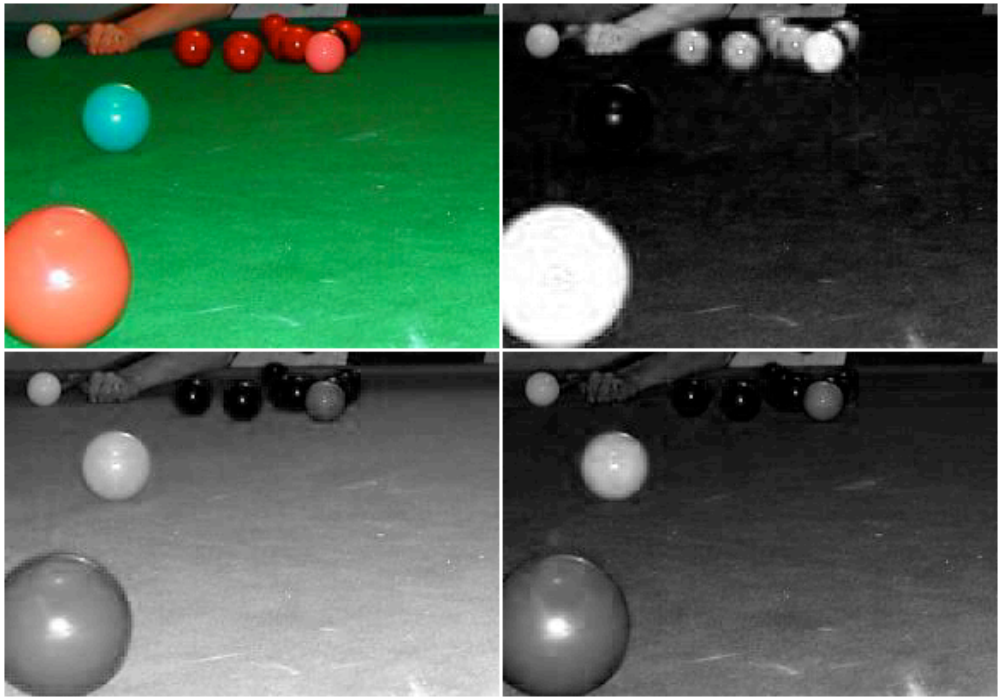
There are many colour representations

RGB	Red, Green, Blue
CMY	Cyan, Magenta, Yellow
YUV	Luminance (Y), Blue minus Luminance (U), Red minus Luminance (V)
YCrCb	Scaled version of YUV
CIE XYZ	Standard reference colour space based on the response of human eye
CIE L*u*V*	Perceptually uniform colour space
CIE L*a*b*	Device independent colour space (all colours perceived by humans)
HSV	Hue, Saturation, Value
HLS	Hue, Luminance, Saturation
HSI	Hue, Saturation, Intensity

Colour Spaces



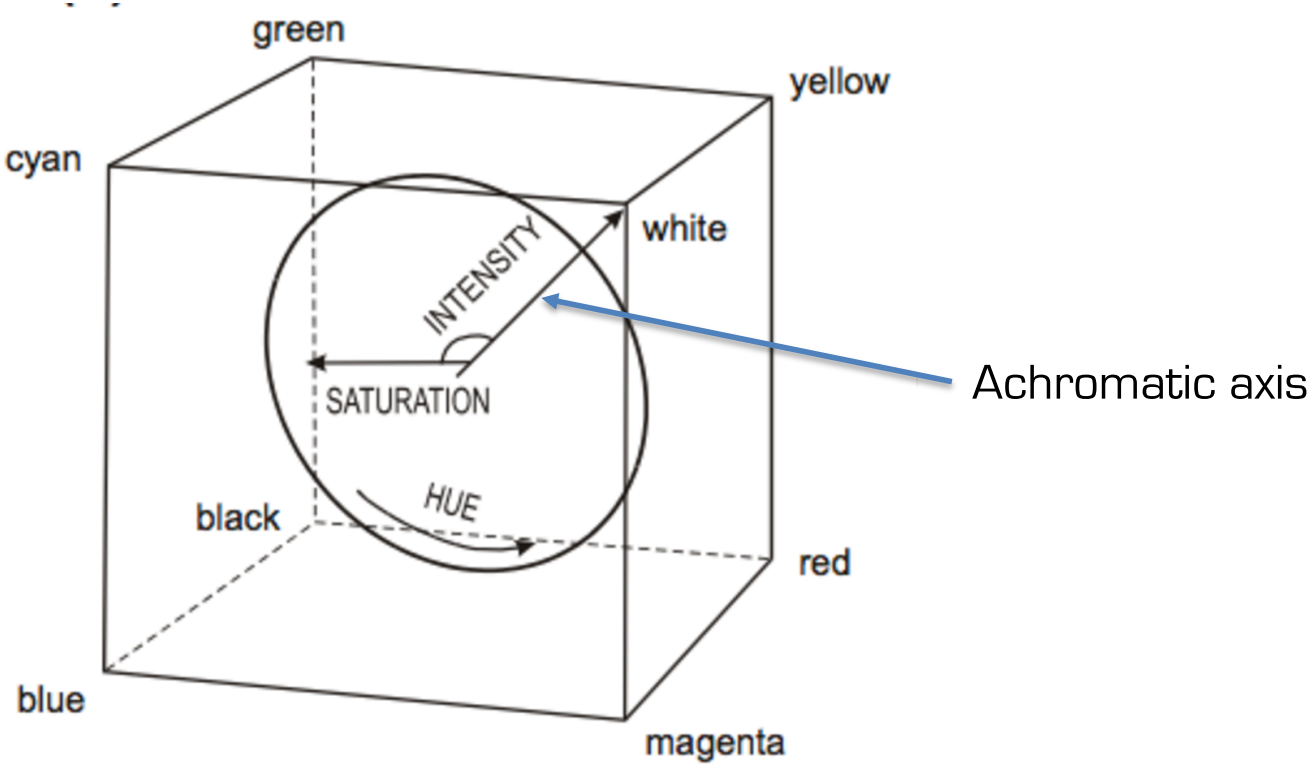
Colour Spaces



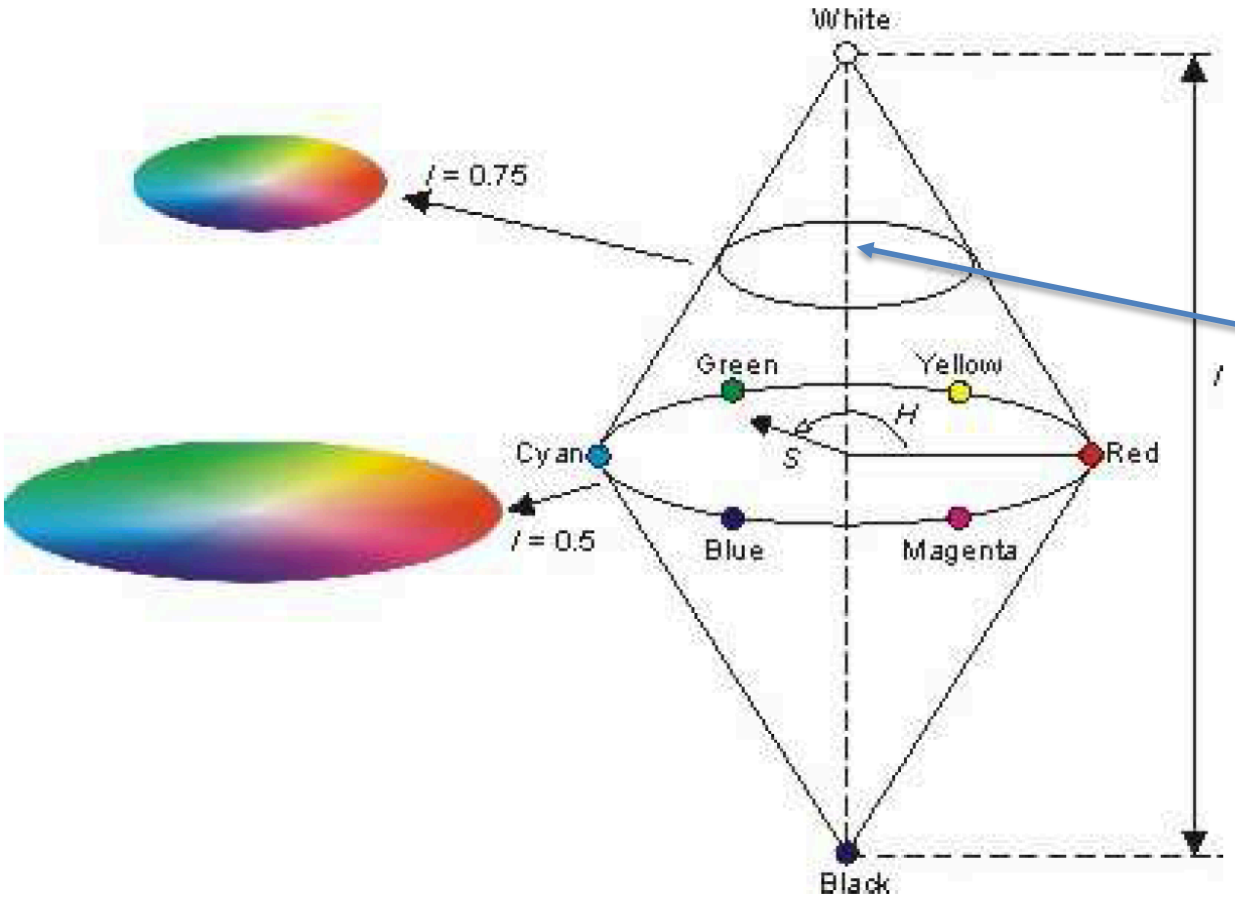
Red Green Blue images

Red (~ 700nm)
Green (~ 546nm)
Blue (~ 436nm)

Colour Spaces

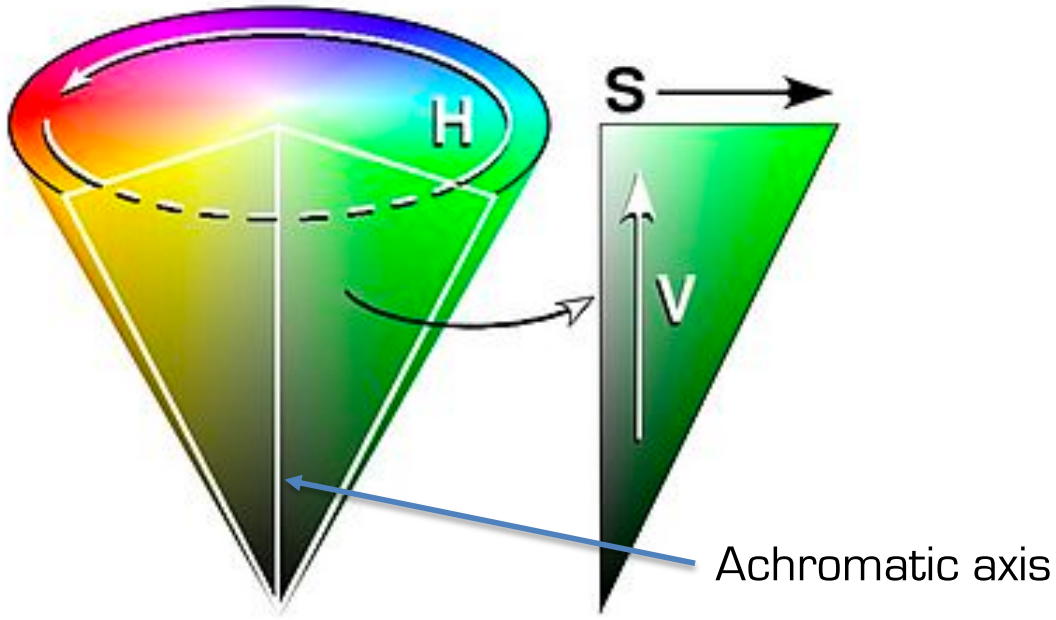


Colour Spaces

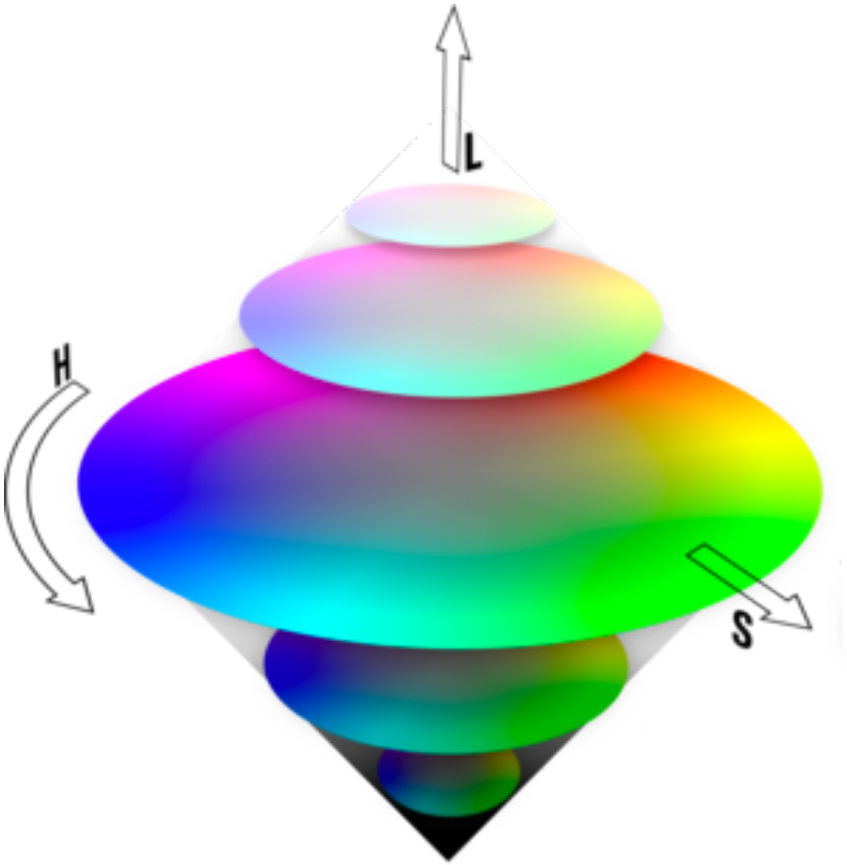


Achromatic axis

Colour Spaces



Colour Spaces



Colour Spaces

Difference between HIS, HLS, and HSV are all based on different definitions of “brightness”

The Generalized Lightness, Hue, and Saturation (GLHS) model defines brightness as follows

$$L(\mathbf{c}) = w_{\min} \cdot \min(\mathbf{c}) + w_{\text{mid}} \cdot \text{mid}(\mathbf{c}) + w_{\max} \cdot \max(\mathbf{c})$$

Where $\min(\mathbf{c})$, $\text{mid}(\mathbf{c})$, and $\max(\mathbf{c})$ return minimum, median, and maximum component of a vector $\mathbf{c}$ in the RGB space

Colour Spaces

w_{min} , w_{mid} , w_{max} determine the colour space

$$w_{min} + w_{mid} + w_{max} = 1, w_{max} > 0$$

HSV $w_{min} = 0, w_{mid} = 0, w_{max} = 1$

HLS $w_{min} = ½ , w_{mid} = 0, w_{max} = = ½$

HIS $w_{min} = 1/3, w_{mid} = 1/3, w_{max} = 1/3$

Colour Spaces

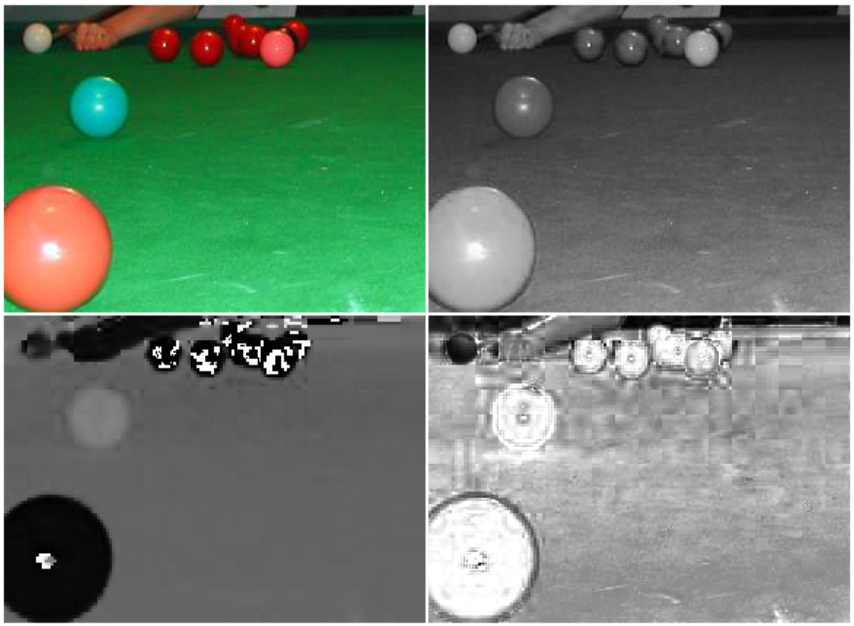
$$I = R + G + B$$

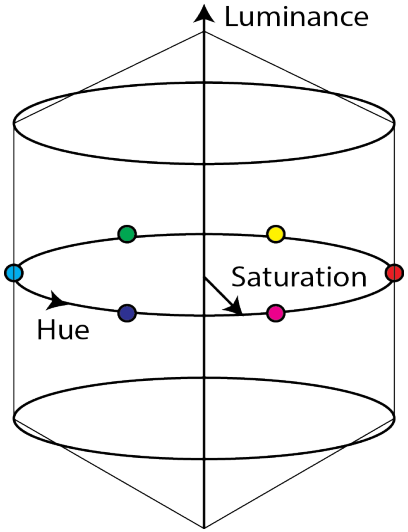
$$h = \arccos\left(\frac{((R-G)+(R-B))}{2\sqrt{(R-G)^2+(R-B)(G-B)}}\right)$$

$$H = \begin{cases} h & \text{if } G \geq B \text{ and not } (R=G=B) \\ 2\pi - h & \text{if } G \leq B \\ \text{undefined} & \text{if } (R=G=B) \end{cases}$$

$$S = 1 - \frac{3\times(\min(R,G,B))}{(R+G+B)}$$

Colour Spaces





Hue Luminance Saturation

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Colour Spaces

Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Demos

The following code is taken from the colourToGreyscale project in the lectures directory of the ACV repository

See:

`colourToGreyscaleImplementation.cpp`

```
/*
  Example use of openCV to convert a colour image to greyscale
  ------------------------------------------------------------
  Implementation file

  David Vernon
  8 May 2017

*/

#include "colourToGreyscale.h"

void colourToGreyscale(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]   = "Input Image";
   char outputWindowName[MAX_STRING_LENGTH]  = "Greyscale Image";

   Mat colourImage;
   Mat greyscaleImage;

   int row;
   int col;
   int channel;
   int temp;

   namedWindow(inputWindowName,   CV_WINDOW_AUTOSIZE);
   namedWindow(outputWindowName,  CV_WINDOW_AUTOSIZE);

   colourImage = imread(filename, CV_LOAD_IMAGE_COLOR);        // Read the file
   // colourImage = imread(filename, CV_LOAD_IMAGE_GRAYSCALE); // just for testing

   printf("number of channels %d\n", colourImage.channels());
```

```
if (!colourImage.data) {                              // Check for invalid input
   printf("Error: failed to read image %s\n",filename);
   prompt_and_exit(-1);
}

printf("Press any key to continue ...\n");

/* test image */
/*
for (row=0; row < colourImage.rows; row++) {
   for (col=0; col < colourImage.cols; col++) {
      for (channel=0; channel < colourImage.channels(); channel++) {
         if (colourImage.channels() == 1) { // failsafe just in case the colourImage is not a colour image or a multichannel image
            colourImage.at<uchar>(row,col) = 80 * (col % 3);             // test image comprises bands of greyscale 0, 80, and 160
         }
         else {
            colourImage.at<Vec3b>(row,col)[channel] = 80 * (col % 3);    // test image comprises bands of greyscale 0, 80, and 160
         }
      }
   }
}
*/
imshow(inputWindowName, colourImage);
```

```
   //CV_Assert(colourImage.type() == CV_8UC3);

   // convert to greyscale by explicit access to colour image pixels
   // we do this simply as an example of one way to access individual pixels
   // see changeQuantisation() for a more efficient method that accesses pixels using pointers

   greyscaleImage.create(colourImage.size(), CV_8UC1);

   for (row=0; row < colourImage.rows; row++) {
      for (col=0; col < colourImage.cols; col++) {
         temp = 0;
         for (channel=0; channel < colourImage.channels(); channel++) {
            if (colourImage.channels()== 1) {
               // failsafe just in case the colour image is not a colour image or a multichannel image
               //temp += colourImage.at<Vec3b>(row,col)[channel]; // don't use this
               temp += colourImage.at<uchar>(row,col);            // use this
            }
            else {
               temp += colourImage.at<Vec3b>(row,col)[channel];
            }
         }
         greyscaleImage.at<uchar>(row,col) = (uchar) (temp / colourImage.channels());
      }
   }

   // alternative ... use OpenCV!!!
   // cvtColor(colourImage, greyscaleImage, CV_BGR2GRAY);

   imshow(outputWindowName, greyscaleImage);

   do{
      waitKey(30);                                  // Must call this to allow openCV to display the images
   } while (!_kbhit());                             // We call it repeatedly to allow the user to move the windows
                                                    // (if we don't the window process hangs when you try to click and drag

   getchar(); // flush the buffer from the keyboard hit

   destroyWindow(inputWindowName);
   destroyWindow(outputWindowName);
}
```

Demos

The following code is taken from the colourToHIS project in the lectures directory of the ACV repository

See:

`colourToHISImplementation.cpp`

```
/*
  Example use of openCV to convert a colour image to hue, intensity, and saturation
  --------------------------------------------------------------------------------
  Implementation file

  David Vernon
  8 May 2017
*/

#include "colourToHIS.h"

void colourToHIS(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]         = "Input Image";
   char hueWindowName[MAX_STRING_LENGTH]           = "Hue Image";
   char intensityWindowName[MAX_STRING_LENGTH]     = "Intensity Image";
   char saturationWindowName[MAX_STRING_LENGTH]    = "Saturation Image";

   Mat colourImage;
   Mat hueImage;
   Mat intensityImage;
   Mat saturationImage;

   int row;
   int col;
   unsigned char red;
   unsigned char green;
   unsigned char blue;
   float hue;
   float saturation;
   float intensity;
```

```
namedWindow(inputWindowName,      CV_WINDOW_AUTOSIZE);
namedWindow(hueWindowName,        CV_WINDOW_AUTOSIZE);
namedWindow(intensityWindowName,  CV_WINDOW_AUTOSIZE);
namedWindow(saturationWindowName, CV_WINDOW_AUTOSIZE);

colourImage = imread(filename, CV_LOAD_IMAGE_COLOR); // Read the file

if (!colourImage.data) {                            // Check for invalid input
   printf("Error: failed to read image %s\n",filename);
   prompt_and_exit(-1);
}

printf("Press any key to continue ...\n");

imshow(inputWindowName, colourImage );

CV_Assert(colourImage.type() == CV_8UC3 );

// convert to HIS by explicit access to colour image pixels
// we do this simply as an example of one way to access individual pixels
// see changeQuantisation() for a more efficient method that accesses pixel using pointers

hueImage.create(colourImage.size(), CV_8UC1);
saturationImage.create(colourImage.size(), CV_8UC1);
intensityImage.create(colourImage.size(), CV_8UC1);
```

```
    intensityImage.create(colourImage.size(), CV_8UC1);

    for (row=0; row < colourImage.rows; row++) {
       for (col=0; col < colourImage.cols; col++) {
          blue  = colourImage.at<Vec3b>(row,col)[0];
          green = colourImage.at<Vec3b>(row,col)[1];
          red   = colourImage.at<Vec3b>(row,col)[2];

          rgb2hsi(red, green, blue, &hue, &saturation, &intensity);
          hueImage.at<uchar>(row,col)        = (char)  (255.0  * (hue/360.0));
          saturationImage.at<uchar>(row,col) = (char)  (saturation * 255);
          intensityImage.at<uchar>(row,col)  = (char)  (intensity * 255);

       }
    }

    imshow(hueWindowName,       hueImage);
    imshow(intensityWindowName, intensityImage);
    imshow(saturationWindowName,saturationImage);

    do{
       waitKey(30);                                  // Must call this to allow openCV to display the images
    } while (!_kbhit());                             // We call it repeatedly to allow the user to move the windows
                                                     // (if we don't the window process hangs when you try to click and drag

    getchar(); // flush the buffer from the keyboard hit

    destroyWindow(inputWindowName);
    destroyWindow(hueWindowName);
    destroyWindow(intensityWindowName);
    destroyWindow(saturationWindowName);
}
```

Demos

The following code is taken from the imageSamplingAndQuantization project in the lectures directory of the ACV repository

See:

```
imageSamplingAndQuantizationApplication.cpp
imageSamplingAndQuantizationImplementation.cpp
```

```
/*
  Example use of openCV to manipulate sampling density and quantization resolution
  by adjusting the size of an image and the number of bits used to represent it
  ------------------------------------------------------------------------------
  Application file

  David Vernon
  8 May 2017

*/

#include "imageSamplingAndQuantization.h"

/* Global variables to allow access by the display window callback function                                    */
/* (we could also collect all parameters together in a structure and pass a pointer to it in the second argument of the callback) */

Mat src;
int scaleFactor             = 1; // default scale factor
int numberOfBits            = 8; // default quantization resolution

char* input_window_name     = "Input Image";
char* resampled_window_name = "Resampled and Requantized Image";
```

```
int main() {

   int end_of_file;
   bool debug = true;
   char filename[MAX_FILENAME_LENGTH];

   int const max_scale_factor     = 16;
   int const max_number_of_bits   = 8;

   FILE *fp_in;

   printf("Example use of openCV to manipulate sampling density and quantization resolution.\n\n");

   if ((fp_in = fopen("../data/imageSamplingAndQuantizationInput.txt","r")) == 0) {
     printf("Error can't open input file imageSamplingAndQuantizationInput.txt\n");
     prompt_and_exit(1);
   }

   namedWindow(input_window_name, CV_WINDOW_AUTOSIZE );     // Create a window for input
   namedWindow(resampled_window_name, CV_WINDOW_AUTOSIZE ); // create a window for output
   resizeWindow(resampled_window_name,0,0);                 // this forces the trackbar to be as small as possible (and to fit in the window)

   scaleFactor             = 1;                             // default scale factor
   numberOfBits            = 8;                             // default quantization resolution

   createTrackbar( "Scale", resampled_window_name, &scaleFactor, max_scale_factor,   processSamplingAndQuantization); // same callback
   createTrackbar( "Bits", resampled_window_name, &numberOfBits, max_number_of_bits, processSamplingAndQuantization); // same callback
```

```
    do {
       end_of_file = fscanf(fp_in, "%s", filename);

       if (end_of_file != EOF) {
          src = imread(filename, CV_LOAD_IMAGE_COLOR);
          if(src.empty()) {
             cout << "can not open " << filename << endl;
             prompt_and_exit(-1);
          }

          printf("Press any key to continue ...\n");

          imshow(input_window_name, src);

          processSamplingAndQuantization(0, 0);

          do{
             waitKey(30);        // Must call this to allow openCV to display the images
          } while (!_kbhit());   // We call it repeatedly to allow the user to move the windows
                                 // (if we don't the window process hangs when you try to click and drag
          getchar();             // flush the buffer from the keyboard hit
       }
    } while (end_of_file != EOF);

    destroyWindow(input_window_name);
    destroyWindow(resampled_window_name);

    fclose(fp_in);

    return 0;
}
```

```
/*
  Example use of openCV to manipulate sampling density and quantization resolution
  by adjusting the size of an image and the number of bits used to represent it
  ------------------------------------------------------------------------------
  Implementation file

  David Vernon
  8 May 2017
*/

/*
 * Elements of this code were derived from software provided
 * as part of "A Practical Introduction to Computer Vision with OpenCV"
 * by Kenneth Dawson-Howe © Wiley & Sons Inc. 2014.  All rights reserved.
 *
 * The relevant code is placed under a similar copyright notice
 *
 */

#include "imageSamplingAndQuantization.h"

/*
 * function processSamplingAndQuantization
 * Trackbar callback - sub-sampling by resizing using integer scale factor input from user
 * Trackbar callback - requantizing using integer number of bits input from user
*/

void processSamplingAndQuantization(int, void*) {

   Mat scaled_image;
   Mat quantized_image;
   Mat resized_image;

   // cvtColor(src, src_grey, CV_BGR2GRAY);

   if (scaleFactor < 1)  // the trackbar has a lower value of 0 which is invalid
      scaleFactor = 1;

   resize(src, scaled_image, Size( src.cols/scaleFactor, src.rows/scaleFactor));

   changeQuantisation(scaled_image, numberOfBits);

   resize(scaled_image, resized_image, src.size(), 0.0, 0.0, INTER_NEAREST);

   imshow(resampled_window_name, resized_image);
 }
```

```
/*
 * This code is provided as part of "A Practical Introduction to Computer Vision with OpenCV"
 * by Kenneth Dawson-Howe © Wiley & Sons Inc. 2014.  All rights reserved.
 */

// This routine is an example of efficient processing of an image.  To do this
// we have to avoid array indexing and instead use pointer arithmetic to work
// through the image values.  We also have to provide separate code for 1 and
// 3 channel images, and have to provide separate code for continuous and padded
// images.  (A padded image is one where there is some unused space at the end
// of each row, typically to align to a Word boundary).
void changeQuantisation(Mat &image, int new_number_of_bits)
{
   if ((new_number_of_bits >= 8) || (new_number_of_bits <= 0))
      return;
   int image_rows = image.rows;
   int image_columns = image.cols;
   int image_channels = image.channels();
   uchar mask = 0xFF << (8-new_number_of_bits); // e.g. if new_number_of_bits=3, mask=0xE0
   if (image.isContinuous()) // i.e. there is no padding at the end of rows
   {
      // Here we process each row of a 1 or 3 channel continuous image.  Hence we
      // can treat the image data values as a single contiguous array.
      uchar* value = image.ptr<uchar>(0);
      uchar* end_value = value + (image_columns*image_rows*image_channels);
      if (image_channels == 1)
         while (value < end_value)
            *value++ = *value & mask;
      else // if (image_channels == 3)
         while (value < end_value)
         {
            // The 3 channel version is more efficient as there are 3 times
            // less comparisons.
            *value++ = *value & mask;
            *value++ = *value & mask;
            *value++ = *value & mask;
         }
   }
```

```
    else if (image_channels == 1)
    {
        // Here we process each row of a 1 channel padded image.  We could use
        // this code for a continuous image but it would be a bit less efficient.
        for (int row=0; row < image_rows; row++) {
            uchar* value = image.ptr<uchar>(row);
            for (int column=0; column < image_columns; column++)
                *value++ = *value & mask;
        }
    }
    else // if (image_channels == 3)
    {
        // Here we process each row of a 3 channel padded image.   We could use
        // this code for a continuous image but it would be a bit less efficient.
        for (int row=0; row < image_rows; row++) {
            uchar* value = image.ptr<uchar>(row);
            for (int column=0; column < image_columns; column++)
            {
                *value++ = *value & mask;
                *value++ = *value & mask;
                *value++ = *value & mask;
            }
        }
    }
}
```

Demos

The following code is taken from the logPolar project in the lectures directory of the ACV repository

See:

`logPolarImplementation.cpp`

```
/*
  Example use of openCV to perform the log-polar transform
  ---------------------------------------------------------
  Implementation file

  David Vernon
  2 May 2017

*/

#include "logPolarTransform.h"

void logPolarTransform(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]            = "Input Image";
   char logPolarWindowName[MAX_STRING_LENGTH]         = "Log-Polar Image";
   char inverseLogPolarWindowName[MAX_STRING_LENGTH]  = "Inverse Log-Polar Image";

   Mat image;
   IplImage *input_image         = NULL;   //  C version  because logpolar is only available in C in 2.10
   IplImage *log_polar_img       = NULL;   //  C version
   IplImage *recovered_log_polar = NULL;   //  C version

   namedWindow(inputWindowName,           CV_WINDOW_AUTOSIZE);
   namedWindow(logPolarWindowName,        CV_WINDOW_AUTOSIZE);
   namedWindow(inverseLogPolarWindowName, CV_WINDOW_AUTOSIZE);

   image = imread(filename, CV_LOAD_IMAGE_COLOR); // Read the file

   if (!image.data) {                             // Check for invalid input
      printf("Error: failed to read image\n");
      prompt_and_exit(-1);
   }
```

```
    printf("Press any key to continue ...\n");

    imshow(inputWindowName, image );                // show our image inside it

    input_image = &image.operator IplImage();

    if (log_polar_img == NULL)        {
       log_polar_img       = cvCreateImage( cvSize(input_image->width,input_image->height), IPL_DEPTH_8U, input_image->nChannels );
       recovered_log_polar = cvCreateImage( cvSize(input_image->width,input_image->height), IPL_DEPTH_8U, input_image->nChannels );
    }

    Point2f center( (float)image.cols / 2, (float)image.rows / 2 );
    double radius = (double)image.cols / 3;
    double M = (double)image.cols / log(radius);

    cvLogPolar(input_image, log_polar_img, center, M, INTER_LINEAR);
    cvLogPolar(log_polar_img, recovered_log_polar, center, M, WARP_INVERSE_MAP + INTER_LINEAR);
    cvShowImage(logPolarWindowName, log_polar_img );
    cvShowImage(inverseLogPolarWindowName, recovered_log_polar );

    do{
       waitKey(30);                                 // Must call this to allow openCV to display the images
    } while (!_kbhit());                            // We call it repeatedly to allow the user to move the windows
                                                    // (if we don't the window process hangs when you try to click and drag

    getchar(); // flush the buffer from the keyboard hit

    destroyWindow(inputWindowName);
    destroyWindow(logPolarWindowName);
    destroyWindow(inverseLogPolarWindowName);
}
```

OpenCV

version 2.4

For documentation on OpenCV data structures, functions, and methods, see

http://docs.opencv.org/2.4/index.html

Reading

R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.

Section 2.3 The digital camera

Section 2.3.1 Sampling and aliasing

Section 2.3.2 Colour

Hanbury, A. “The Taming of the Hue, Saturation, and Brightness Colour Space”, Proc. Computer Vision Winter Workshop (CVWW), Austria, 2002.