

# Module 1

## Introduction

- **Quickstart: Read & Display Images**
- **Introduction**
- **Types of Images**
- **Imaging Geometry**
- **Image Representation**
- **Perceptrons**

# Read & Display “.bin” Images

- Many of our images will have file type “.bin”
- This is not a “standard” file type!
  - ▶ These files contains raw image data without any header information.
- Programs like *Microsoft Office Picture Manager* can't display “.bin” files directly.
- But raw image files are easy to work with because the format is simple and there's no complicated header to parse.

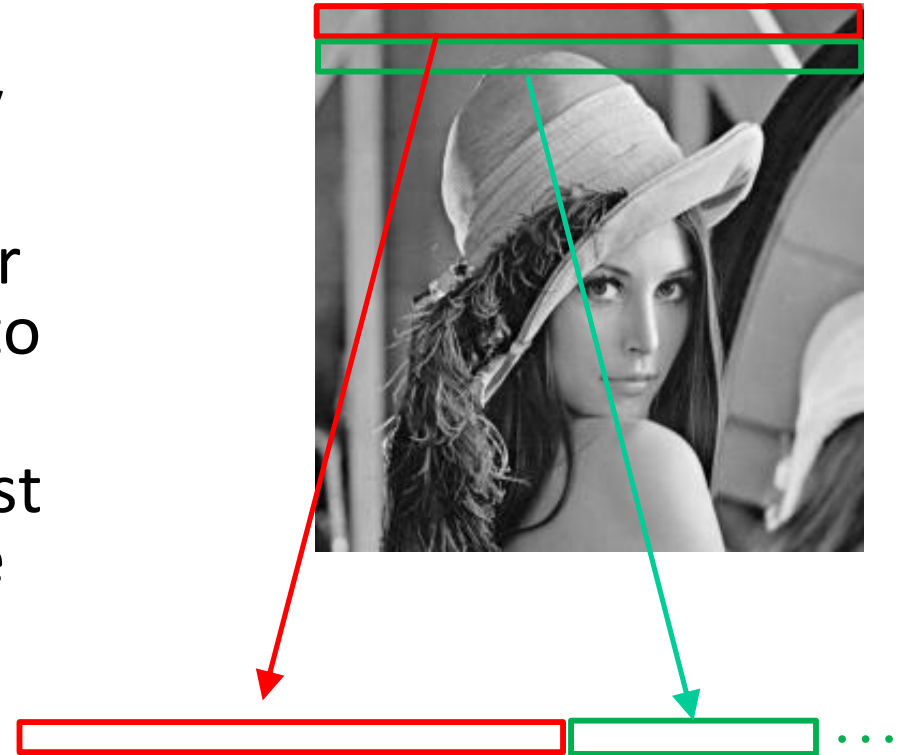
# What's Inside a ".bin" Image File

- It is usually a grayscale image.
- Each pixel (picture element) is one byte.
  - ▶ The data type is uchar (also known as uint8).
  - ▶ The min value is 00000000b = 00h = 0d, which displays as the darkest black.
  - ▶ The max value is 11111111b = FFh = 255d, which displays as the brightest white.



# Inside a “.bin” Image File

- The file contains a linear array of bytes: one for each pixel.
- The pixels are stored in “raster scan” order: left to right, top to bottom.
- For a  $256 \times 256$  image, the first 256 bytes are the pixels of the top row from left to right.
- The next 256 bytes are the pixels of the second row, from left to right, and so on.



# Read & Display “.bin” Using Matlab

```
fidLena = fopen('lena.bin','r');  
[Lena,junk] = fread(fidLena,[256,256],'uchar');  
junk % echo the number of bytes that were read  
Lena = Lena'; % you must transpose the image
```

```
figure(1);colormap(gray(256));  
image(Lena);  
axis('image');  
title('Original Lena Image');  
print -dtiff M_Lena.tif; % write figure as tif
```

```
fidOut = fopen('Outfile.bin','w+');  
LenaOut = Lena'; % transpose before writing  
fwrite(fidOut,LenaOut,'uchar'); % write raw image data  
fclose(fidLena);fclose(fidOut);
```

# Some Notes About Matlab

- When you use `fread` to read a “.bin” image, you must transpose the array after reading.
  - ▶ This is a legacy issue related to the fact that Matlab was originally written in FORTRAN, but was later re-implemented in C.
  - ▶ In FORTRAN, 2D arrays are stored in memory in column-major order, but in C they are stored in row-major order (see next page).
- If you read the image into a 2D Matlab array `X` (and transpose), then `X(m,n)` is the pixel at row=`m` and col=`n`. `X(1,1)` is the upper left pixel of the image. `X(5,7)` is the pixel on row 5 and column 7.
- Reasons TO use Matlab: it's easy and has lots of built in function support (to display images, for example).
- Reasons NOT to use Matlab: it can be slow for big images!
- Use matrix operations and avoid loops wherever possible!
  - ▶ Ex: to copy a block of pixels, use `K(1:256,1:128) = J(1:256,129:256)` instead of a loop.

# More on 2D Array Indexing

- A 2D C array is stored in memory in row-major order. If you traverse the elements in order, the **last** index varies fastest. To avoid cache faults when accessing  $x[m][n]$  with nested loops, the outer loop should be  $m$  and the inner loop should be  $n$ .
- A FORTRAN 2D array is stored in column-major order. If you traverse the elements in order, the **first** index varies fastest. To avoid cache faults when accessing  $x(m,n)$  with nested loops, the outer loop should be  $n$  and the inner loop should be  $m$ .
- When using large 2D arrays in Matlab, it is important to remember that Matlab was originally written in FORTRAN.

```
%  
% LoopTime.m  
%  
xsize = 20000;  
x = rand(xsize,xsize);  
y = zeros(xsize,xsize);  
tic  
for row=1:xsize  
    for col=1:xsize  
        y(row,col) = x(row,col);  
    end  
end  
toc  
tic  
for col=1:xsize  
    for row=1:xsize  
        y(row,col) = x(row,col);  
    end  
end  
toc
```

## Matlab Console Output:

```
>> LoopTime  
Elapsed time is 28.483111 seconds.  
Elapsed time is 2.889330 seconds.
```

- ▶ The loops run 10x faster the 2<sup>nd</sup> way!

# Read & Display “.bin” Using C

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

#define BYTE unsigned char

/*
 * Function Prototypes (forward declarations)
 */
void disk2byte();
void byte2disk();

/*-----*/
/*  MAIN                                     */
/*-----*/

main(argc,argv)

    int    argc;
    char   *argv[];
{

    int    size;           /* num rows/cols in images */
    int    so2;           /* size / 2 */
    int    i;             /* counter */
    int    row;           /* image row counter */
    int    col;           /* image col counter */
    BYTE   *I1;           /* input image I1 */
```

```

BYTE    *I2;           /* input image I2 */
BYTE    *J;           /* output image J */
BYTE    *K;           /* output image K */
char    *InFn1;       /* input filename for image I1 */
char    *InFn2;       /* input filename for image I2 */
char    *OutFnJ;      /* output filename for image J */
char    *OutFnK;      /* output filename for image K */

/*
 * Check for proper invocation, parse args
 */
if (argc != 6) {
    printf("\n%s: Swap image halves for hw01.",argv[0]);
    printf("\nUsage: %s size InFn1 InFn2 OutFnJ OutFnK\n",
        argv[0]);
    exit(0);
}
size = atoi(argv[1]);
if (size % 2) {
    printf("\n%s: size must be divisible by 2.\n",argv[0]);
    exit(0);
}
InFn1 = argv[2];
InFn2 = argv[3];
OutFnJ = argv[4];
OutFnK = argv[5];

so2 = size >> 1;

/*
 * Allocate image arrays
 */
if ((I1 = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
}

```

```

if ((I2 = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
}
if ((J = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
}
if ((K = (BYTE *)malloc(size*size*sizeof(BYTE))) == NULL) {
    printf("\n%s: free store exhausted.\n",argv[0]);
    exit(-1);
}

/*
 * Read input images
 */
disk2byte(I1,size,size,InFn1);
disk2byte(I2,size,size,InFn2);

/*
 * Make output image J: left half is I1, right half is I2
 */
for (i=row=0; row < size; row++) {
    for (col=0; col < so2; col++,i++) {
        J[i] = I1[i];
    }
    for ( ; col < size; col++,i++) {
        J[i] = I2[i];
    }
}

```

```

/*
 * Make output image K: swap left and right halves of J
 */
for (row=0; row < size; row++) {
    for (col=0; col < so2; col++) {
        K[row*size + col] = J[row*size + col+so2];
    }
    for ( ; col < size; col++) {
        K[row*size + col] = J[row*size + col-so2];
    }
}

/*
 * Write the output images
 */
byte2disk(J,size,size,OutFnJ);
byte2disk(K,size,size,OutFnK);

return;
} /*----- Main -----*/

/*-----
 * disk2byte.c
 *
 * function reads an unsigned char (byte) image from disk
 *
 *
 * jph 15 June 1992
 *
-----*/

void disk2byte(x,row_dim,col_dim,fn)

    BYTE    *x;           /* image to be read */
    int     row_dim;      /* row dimension of x */
    int     col_dim;      /* col dimension of x */
    char    *fn;          /* filename */
{

```

```

int fd;          /* file descriptor */
int n_bytes;    /* number of bytes to read */

/*
 * detect zero dimension input
 */
if ((row_dim==0) || (col_dim==0)) return;

/*
 * open the file
 */
if ((fd = open(fn, O_RDONLY))== -1) {
    printf("\ndisk2byte.c : could not open %s !",fn);
    return;
}

/*
 * read image data from the file
 */
n_bytes = row_dim * col_dim * sizeof(unsigned char);
if (read(fd,x,n_bytes) != n_bytes) {
    printf("\ndisk2byte.c : complete read of %s did not succeed.",fn);
}

/*
 * close file and return
 */
if (close(fd) == -1) printf("\ndisk2byte.c : error closing %s.",fn);
return;
}

```

```

/*-----
 * byte2disk.c
 *
 * function writes an unsigned char (byte) image to disk
 *
 *
 * jph 15 June 1992
 *
-----*/

```

```
void byte2disk(x,row_dim,col_dim,fn)
```

```

    BYTE *x;           /* image to be written */
    int  row_dim;      /* row dimension of x */
    int  col_dim;      /* col dimension of x */
    char *fn;          /* filename */
{
    int fd;            /* file descriptor */
    int n_bytes;       /* number of bytes to read */

    /*
     * detect zero dimension input
     */
    if ((row_dim==0) || (col_dim==0)) return;

    /*
     * create and open the file
     */
    if ((fd = open(fn, O_WRONLY | O_CREAT | O_TRUNC, 0644))== -1) {
        printf("\nbyte2disk.c : could not open %s !",fn);
        return;
    }
}

```

```
/*
 * write image data to the file
 */
n_bytes = row_dim * col_dim * sizeof(unsigned char);
if (write(fd,x,n_bytes) != n_bytes) {
    printf("\nbyte2disk.c : complete write of %s did not succeed.",fn);
}

/*
 * close file and return
 */
if (close(fd) == -1) printf("\nbyte2disk.c : error closing %s.",fn);
return;
}
```

# Some Notes About C

- C is good because it is FAST and has powerful syntax for performing low level (bit & byte) operations on image data.
- You need a good compiler. See “links” on the course web site to obtain the excellent GNU gcc compiler. It will compile the code in the previous example.
- You need a good debugger. See “links” on the course web site to get DDD.
- You need good library routines to display images and to read images with headers. See “links” on the course web site to get ImageMagick.
  - ▶ Also see the “Image Magick HOWTO” under “Handouts” on the course web site.
- Array indexing in C starts at ZERO. This is an important difference from Matlab.
  - ▶ The top row of the image is row 0. The first column is col 0.
  - ▶ For a  $256 \times 256$  image,  $X[m*256 + n]$  is the pixel at row=m and col=n.
  - ▶ For a 2D C array,  $X[m][n]$  is the pixel at row=m and col=n.
- If you read the image into a 1D C array, it’s easy to handle the case where the image SIZE is unknown at compile time.
- The main reasons to use C are that it is FAST and PORTABLE... faster debugging and doesn’t depend on having a Matlab installation.

# Convert “.bin” Image to “.pgm”

- If you want to directly display a “.bin” image using standard programs like *Microsoft Office Picture Manager*, you need to add a header.
- The easiest way is to make a “.pgm” file.
- For a  $256 \times 256$  “.bin” image, add the following 15-byte header:

HEX	50	35	0A	32	35	36	20	32
ASCII	P	5	.	2	5	6	_	2

HEX	35	36	0A	32	35	35	0A
ASCII	5	6	.	2	5	5	.

- You can do this manually with a hex-capable editor like vi (unix) or Vim (windows or unix).
- Or you can use a program like the function “pgm\_convert.m” available on the course web site under “Code.”

# Course Objectives

- Learn **Digital Image & Video Processing**
  - ▶ Theory
  - ▶ Algorithms and Programming
  - ▶ Applications and Projects
- Have fun doing it

# Some Good Books

- ***Digital Image Processing***, R.C. Gonzalez and R. Woods, 3rd Edition, 2012.  
User-friendly textbook, nicely illustrated. Used previously in this course
- ***Digital Image Processing***, W.K. Pratt, Wiley, 4th Edition, 2007.  
Encyclopedic, rather dated.
- ***The Essential Guide to Image Processing***, Al Bovik, ed., Academic Press, 2009.  
Significant overlap with the notes, examples in Labview.
- ***Fundamentals of Digital Image Processing***, Jain, 1988.  
Handbook-style, terse. Meant for advanced level.
- ***Digital Video Processing***, M. Tekalp, Prentice-Hall, 1995.  
Only book devoted to digital video; high-level; excellent
- ***Machine Vision***, Jain, Kasturi, and Schunk, McGraw-Hill, 1995  
Beginner's book on computer vision.
- ***Robot Vision***, B.K.P. Horn, MIT Press, 1986  
Advanced-level book on computer vision.

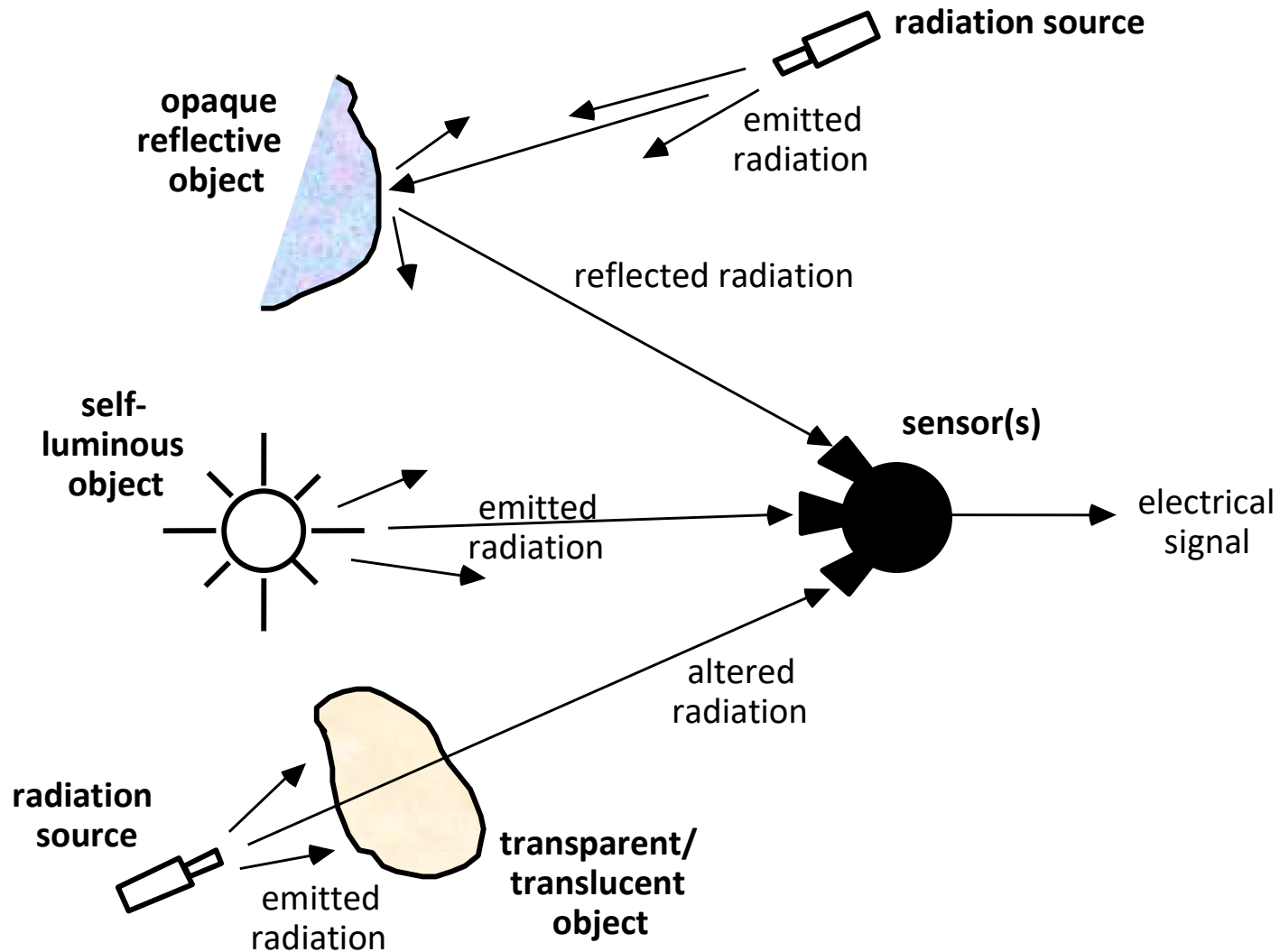
# Journals

- *IEEE Transactions on:*
  - *Image Processing*
  - *Pattern Analysis & Machine Intelligence*
  - *Multimedia*
  - *Geoscience and Remote Sensing*
  - *Medical Imaging*
- *Computer Vision, Graphics, and Image Processing*
  - *Image Understanding*
  - *Graphics and Image Processing*
- *Pattern Recognition*
- *Image and Vision Computing*
- *Journal of Visual Communication and Image Representation*

# Other Types of Pictures

- In **this class**, we focus on **visible-light photographic images** – so much material to cover!
- But images from **everywhere** on the **electromagnetic band** can be obtained!
- **Thermal (IR), gamma-ray, X-ray, UV**, etc....
- Non-EM modalities such as **ultrasound, magnetic resonances, positron emissions**, and much more.
- **All are captured images of the natural world** by sensing **natural radiation**.
- The **techniques** and **models** we **study here** usually have **analogous models** in other **sensing domains**.

# Three Types of Images



# Type #1: Reflection Images

- Image information is **surface** information: how an object **reflects/absorbs** radiation
  - Optical (visual, photographic)
  - Radar
  - Ultrasound, sonar (non-EM)
  - Electron Microscopy

# Type #2: Emission Images

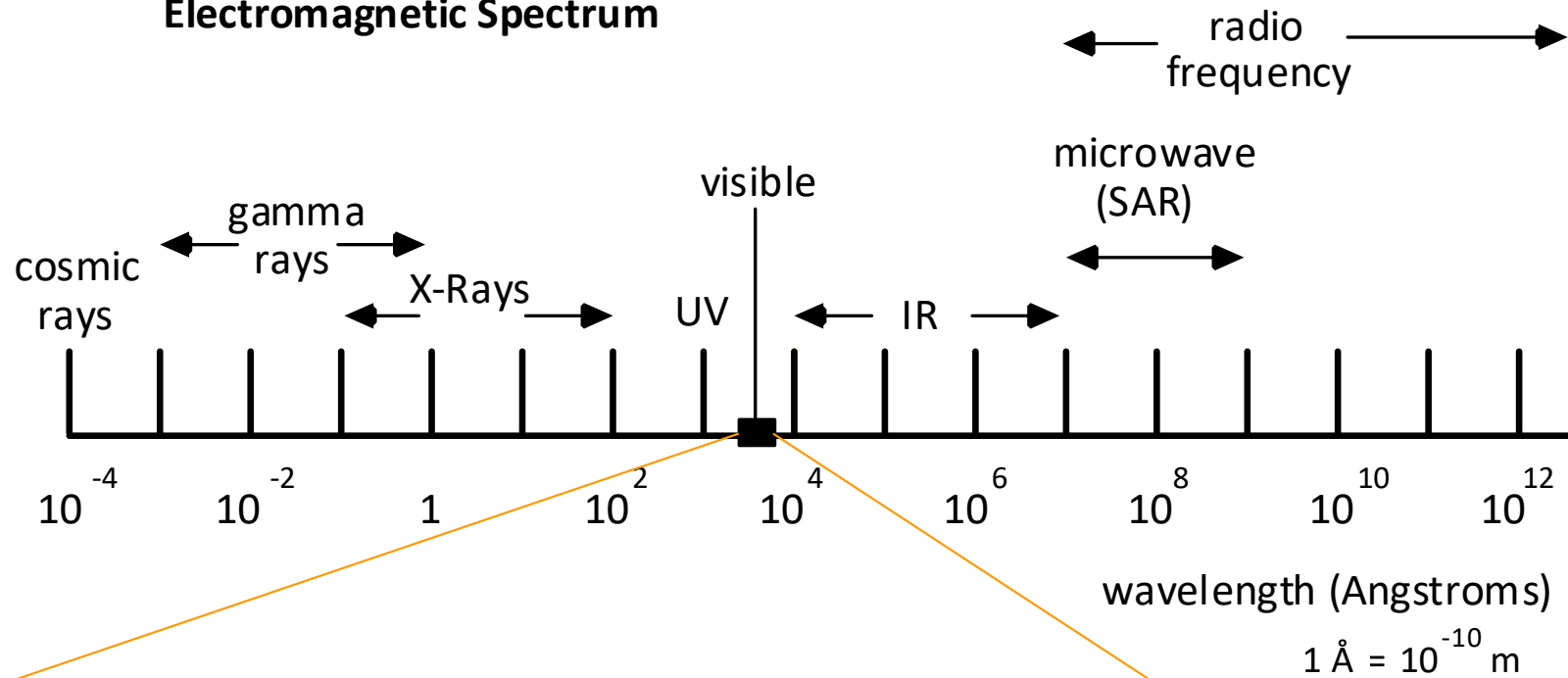
- Image information is **internal** information:  
how an object **creates** radiation
  - Thermal, infrared (FLIR)
  - Astronomy (stars, nebulae, etc.)
  - Nuclear (particle emission, e.g., MRI)

# Type #3: Absorption Images

- Image information is **internal** information: how an object **modifies/absorbs** radiation
  - X-Rays in many applications
  - Brightfield optical microscopy
  - Tomography (CAT, PET) in medicine
  - “Vibro-Seis” in geophysical prospecting

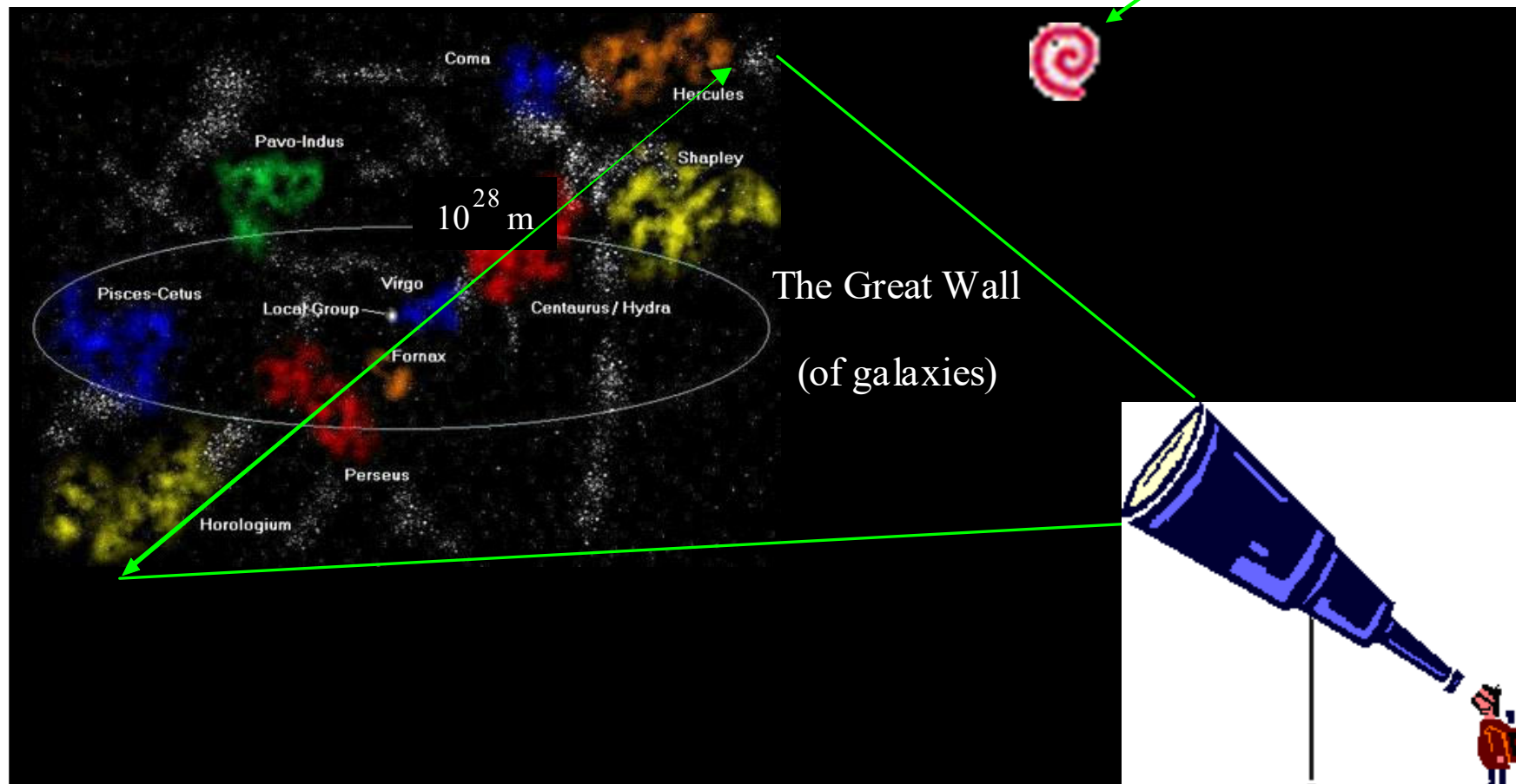
# Electromagnetic Radiation

Electromagnetic Spectrum



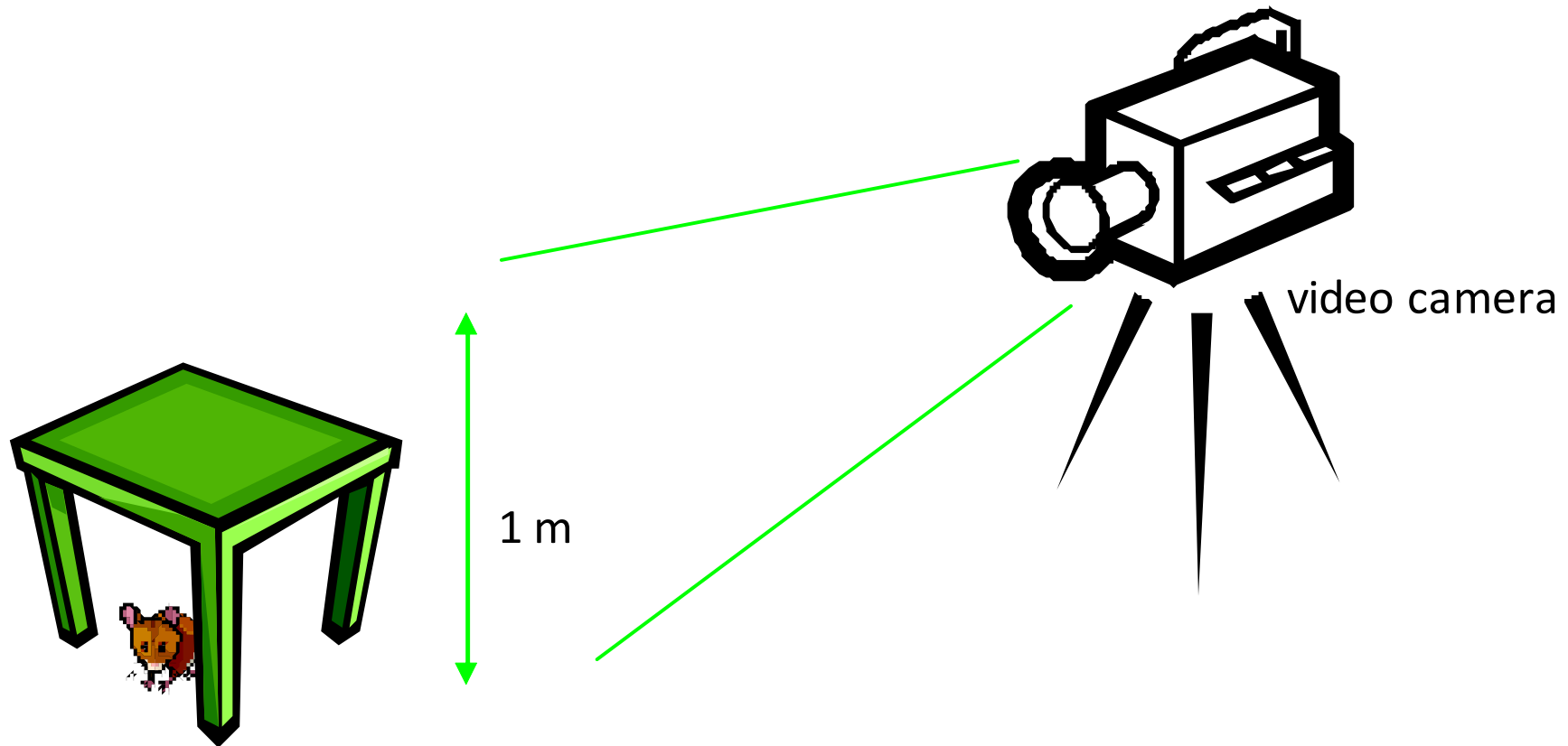
# Scales of Imaging

From the gigantic...



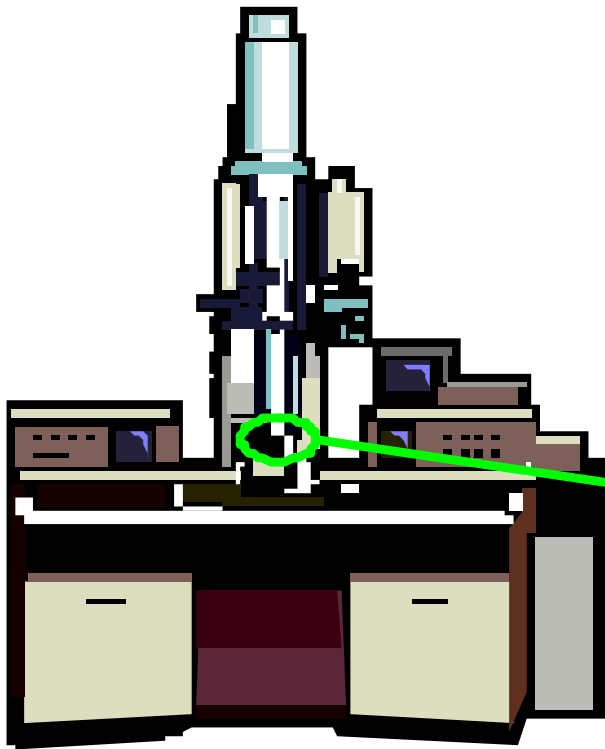
# Scales of Imaging

...to the everyday ...

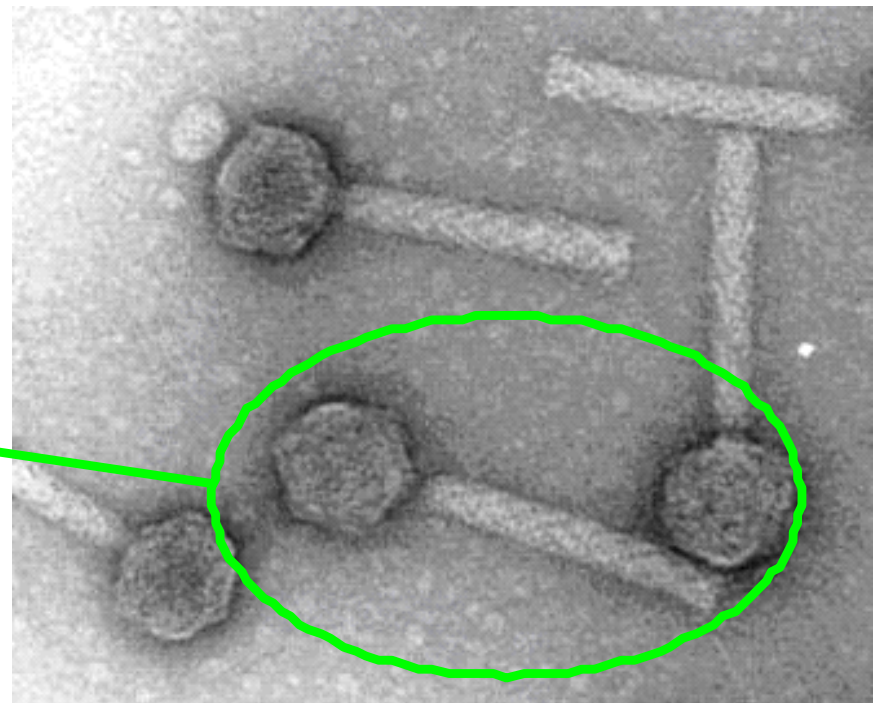


# Scales of Imaging

...to the **tiny**.



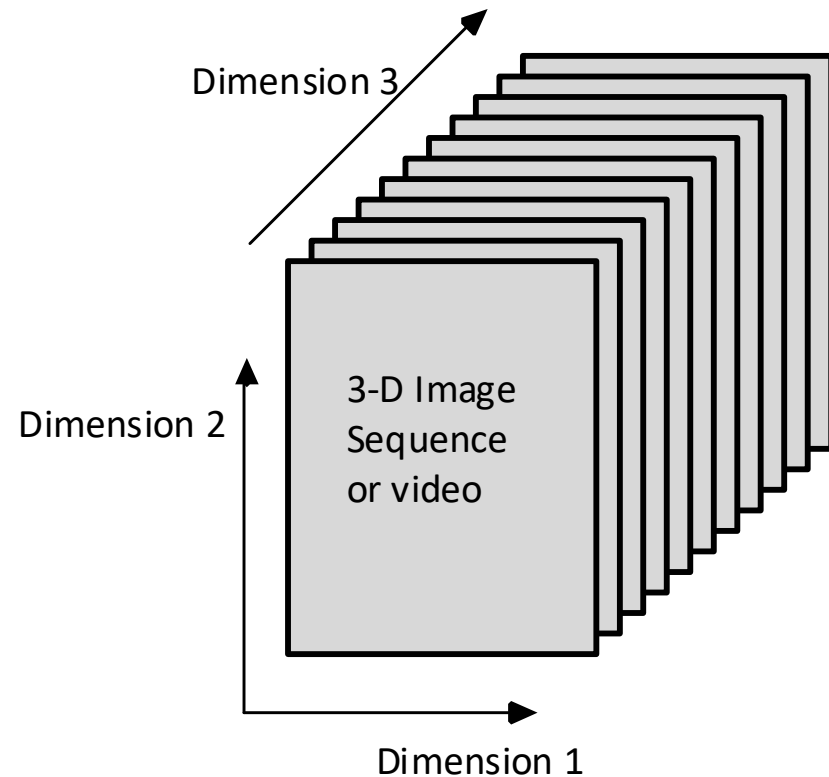
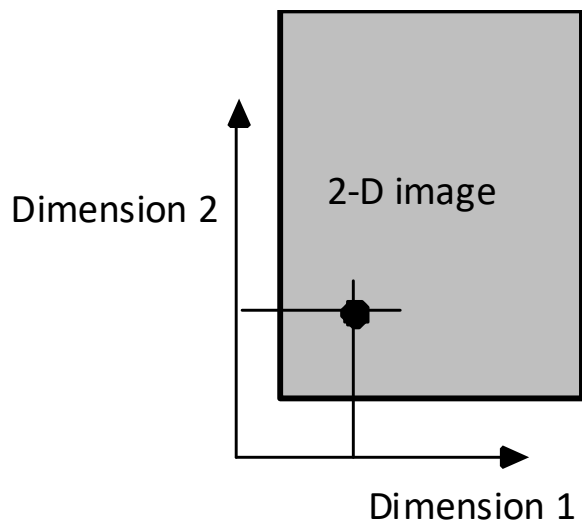
electron microscope



$10^{-6}$  m

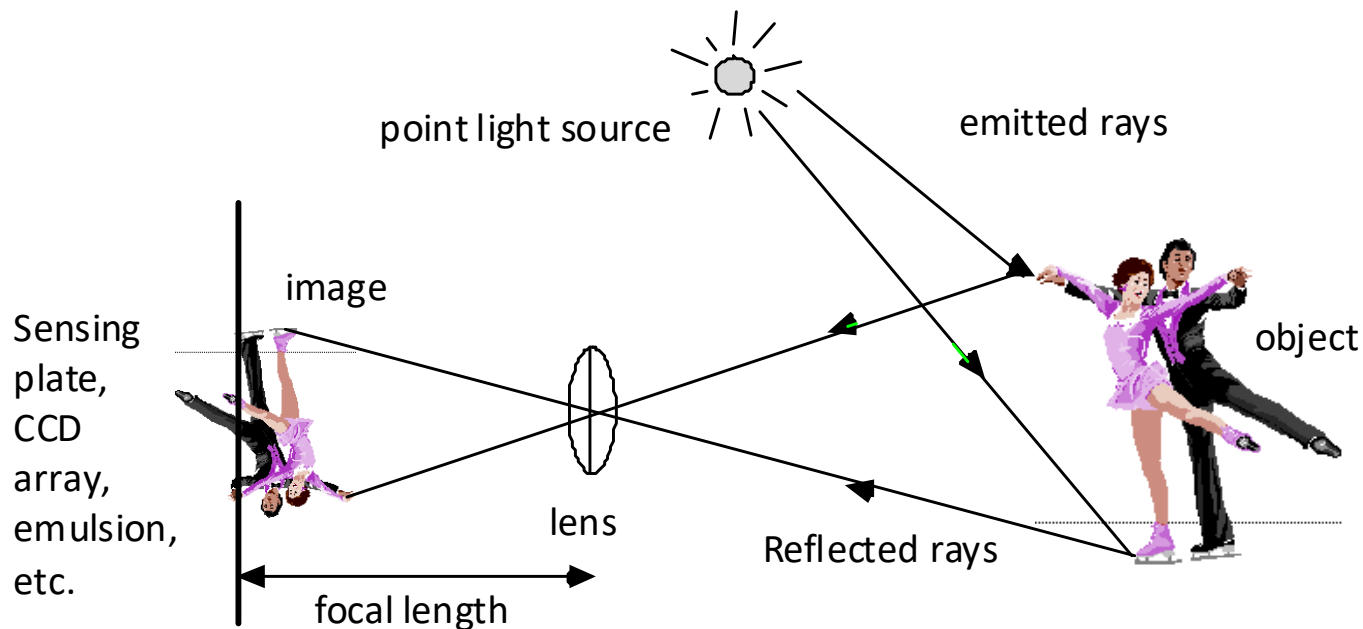
# Dimensionality of Images

- Images and videos are **multi-dimensional** ( $\geq 2$  dimensions) signals.



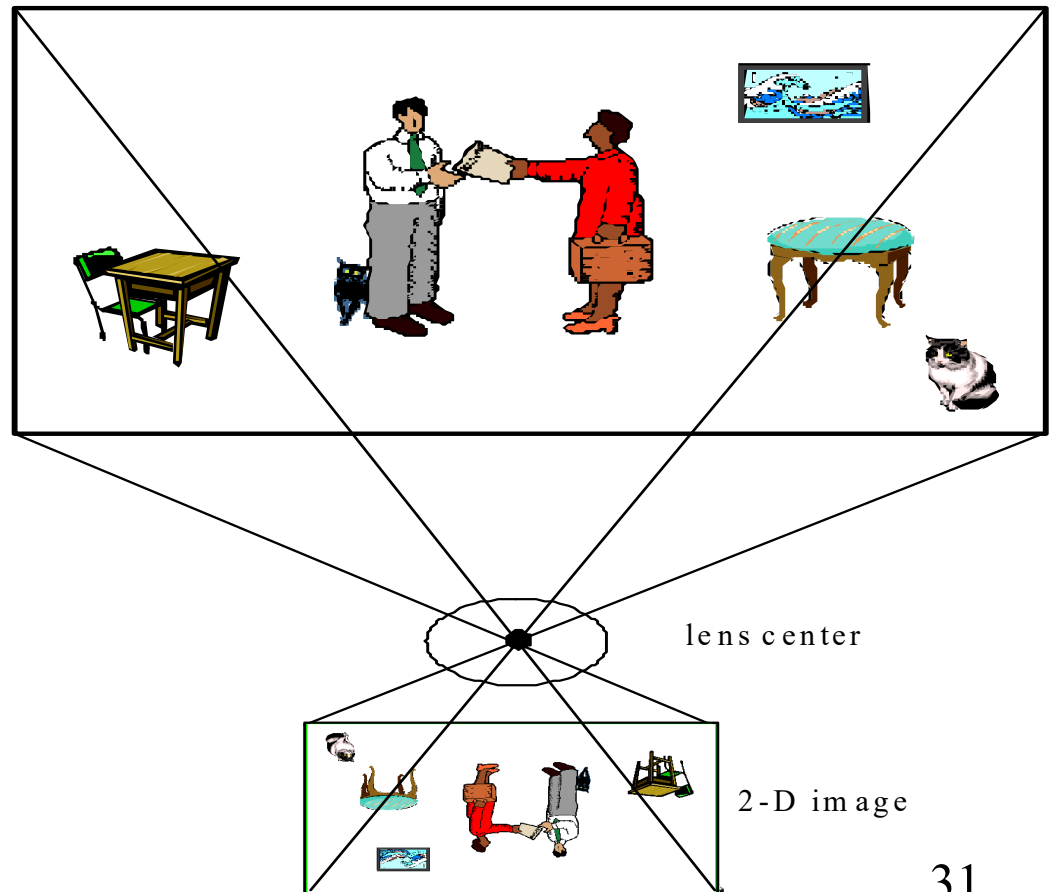
# Optical Imaging Geometry

- Assume **reflection imaging** with visible light.
- Let's quantify the **geometric relationship** between 3-D world coordinates and projected 2-D image coordinates.



# 3D-to-2D Projection

- Image projection is a **reduction of dimension** (3D-to-2D): 3-D info is **lost**. Getting this info back is **very hard**.



- It is a topic of many years of intensive research: “Computer Vision”

# Perspective Projection

- There is a geometric relationship between **3-D space coordinates** and **2-D image coordinates** under perspective projection.
- We will require some **coordinate systems**:

# Projective Coordinate Systems

## Real-World Coordinates

- $(X, Y, Z)$  denote points in 3-D space
- The **origin**  $(X, Y, Z) = (0, 0, 0)$  is the **lens center**

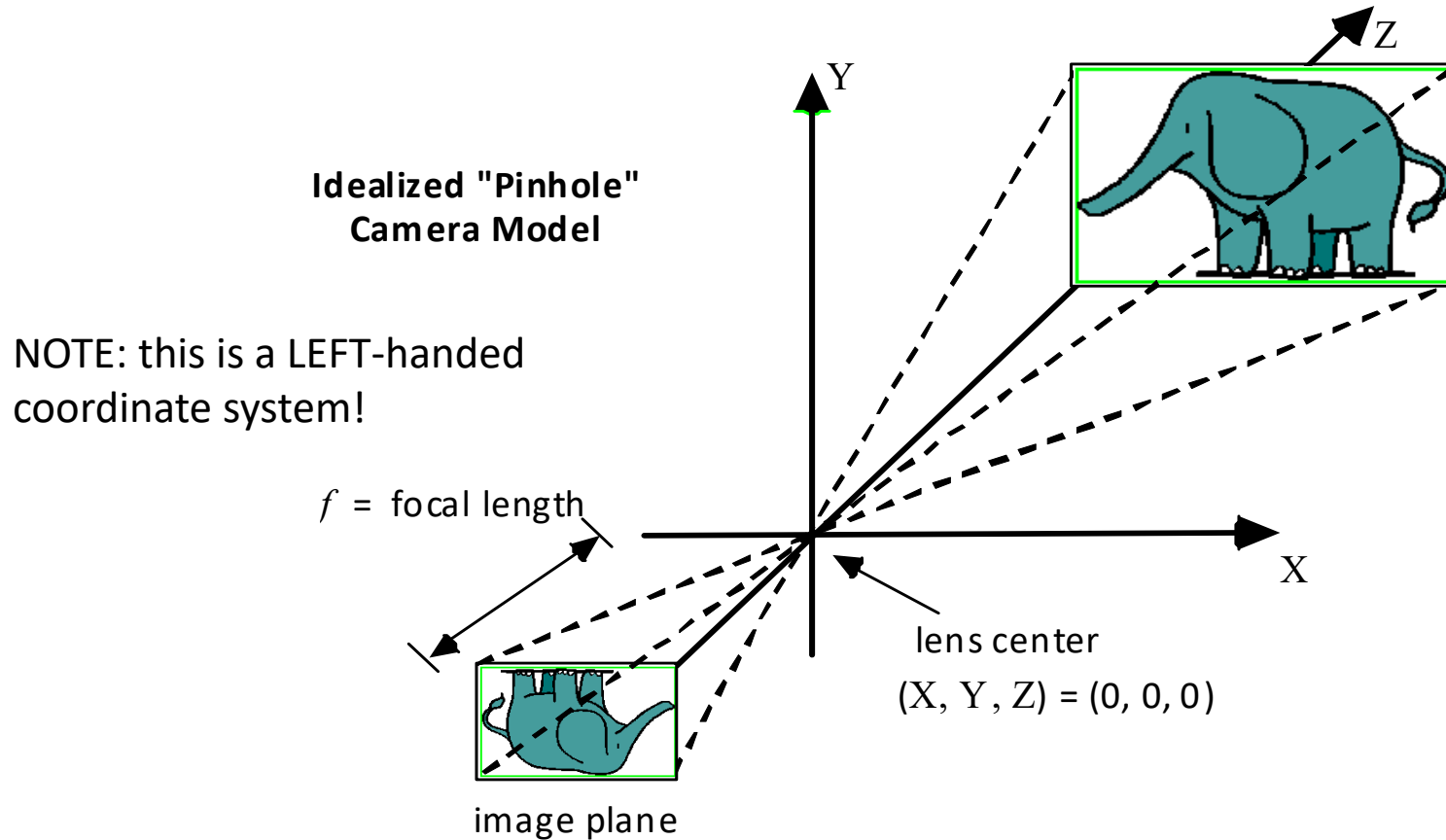
## Image Coordinates

- $(x, y)$  denote points in the 2-D image
- The  $x - y$  plane is chosen parallel to the  $X - Y$  plane
- The **optical axis** passes through both origins

# Pinhole Projection Geometry

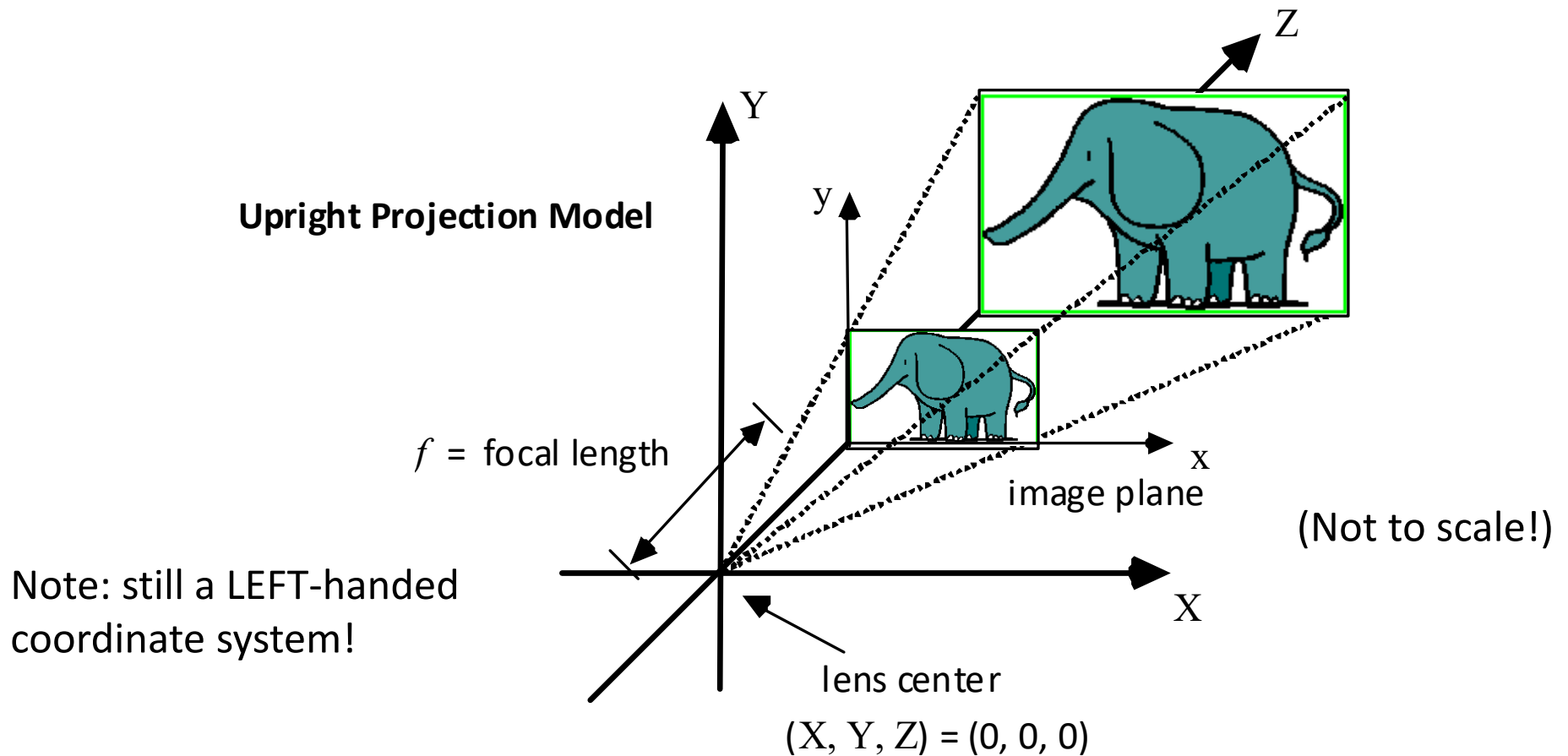
- The lens is modeled as a **pinhole** through which all light rays hitting the image plane pass.
- The image plane is one **focal length  $f$**  from the lens. This is where the camera is in focus.
- The image is **recorded** at the image plane, using a photographic emulsion, CCD sensor, etc.

# Pinhole Projection Geometry

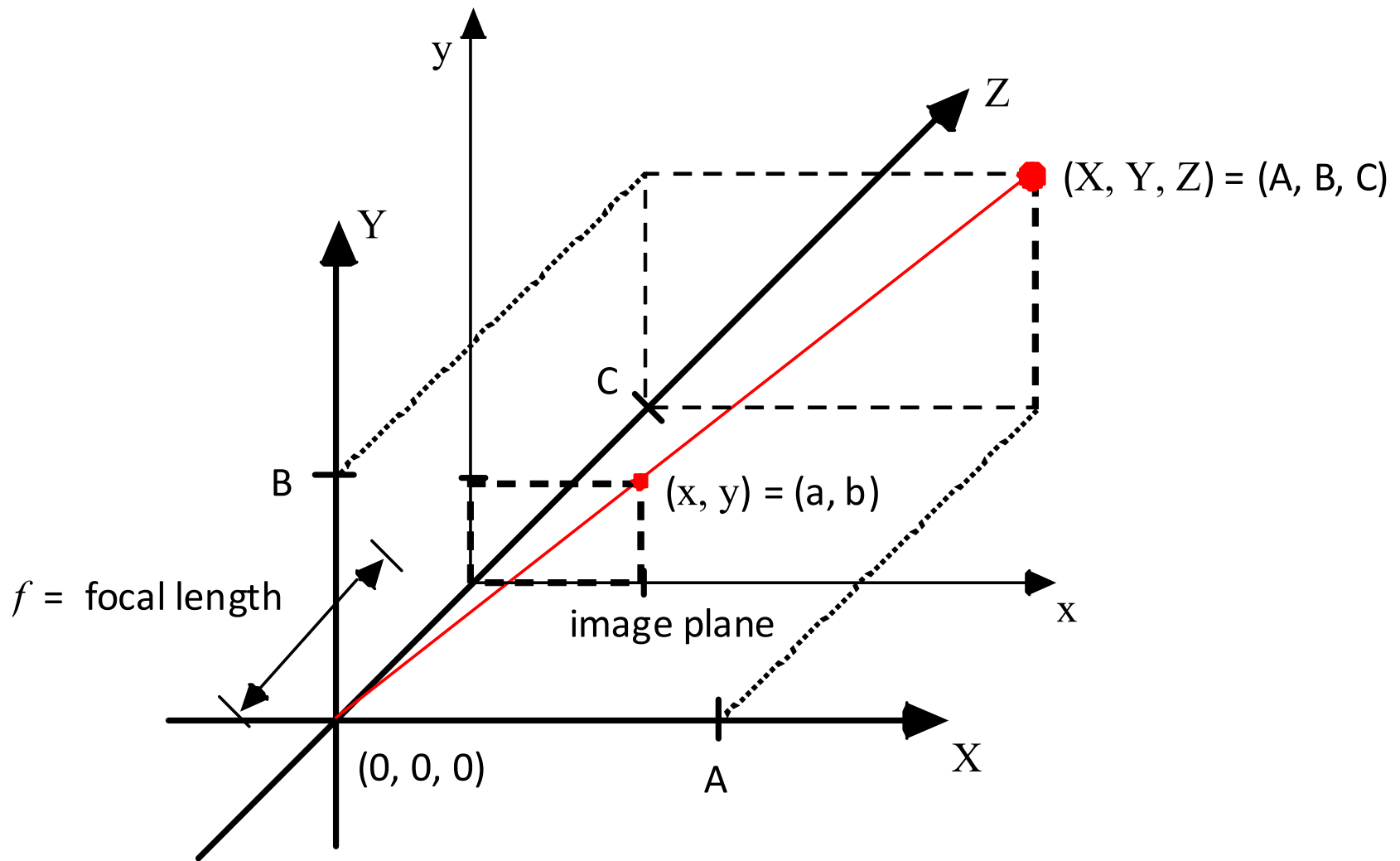


**Problem:** In this model (and in reality), the image is reversed and upside down. It is convenient to change the model to correct this.

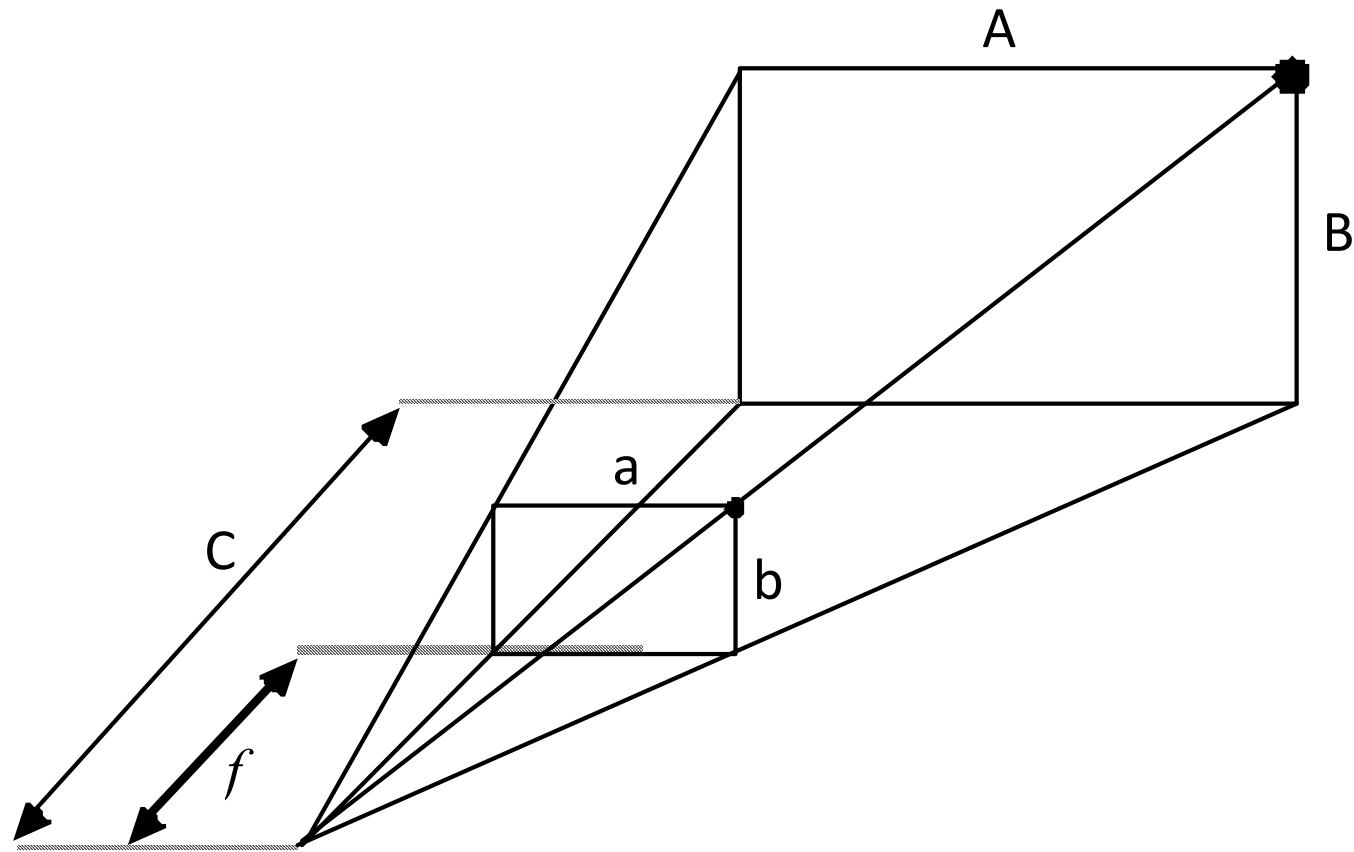
# Upright Projection Geometry



- Let us make our model more mathematical...



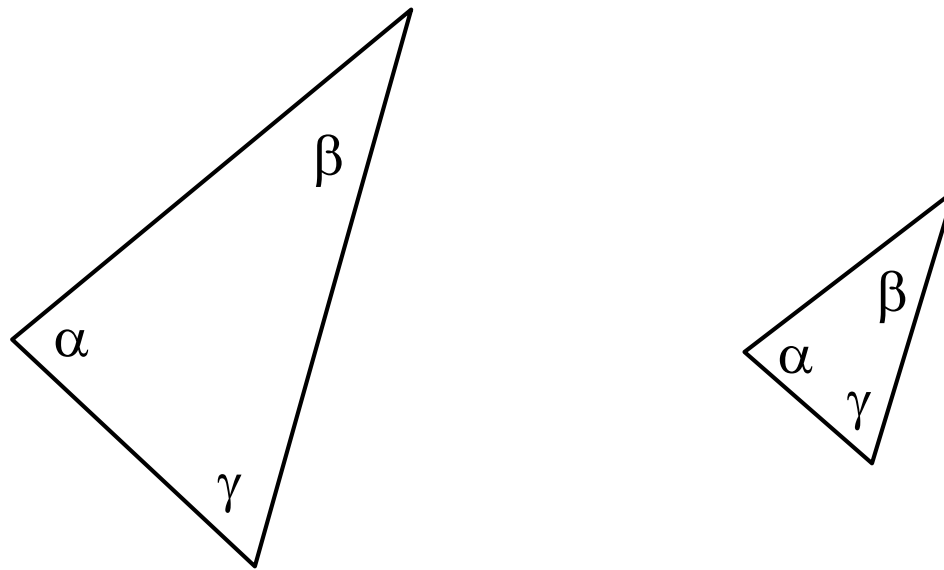
- All of the relevant coordinate axes and labels ...



- This equivalent **simplified diagram** shows only the relevant data relating  $(X, Y, Z) = (A, B, C)$  to its projection  $(x, y) = (a, b)$ .

# Similar Triangles

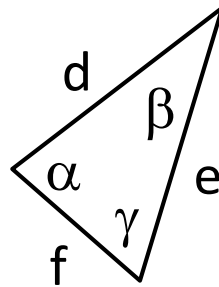
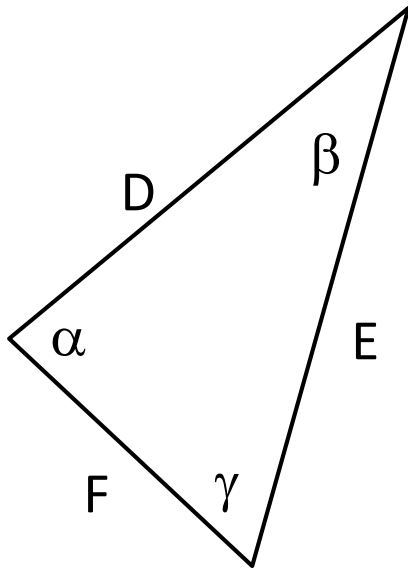
- Triangles are **similar** if their **corresponding angles are equal**:



**similar triangles**

# Similar Triangles Theorem

- Similar triangles have their side lengths in the **same proportions**.



$$\frac{D}{E} = \frac{d}{e}$$

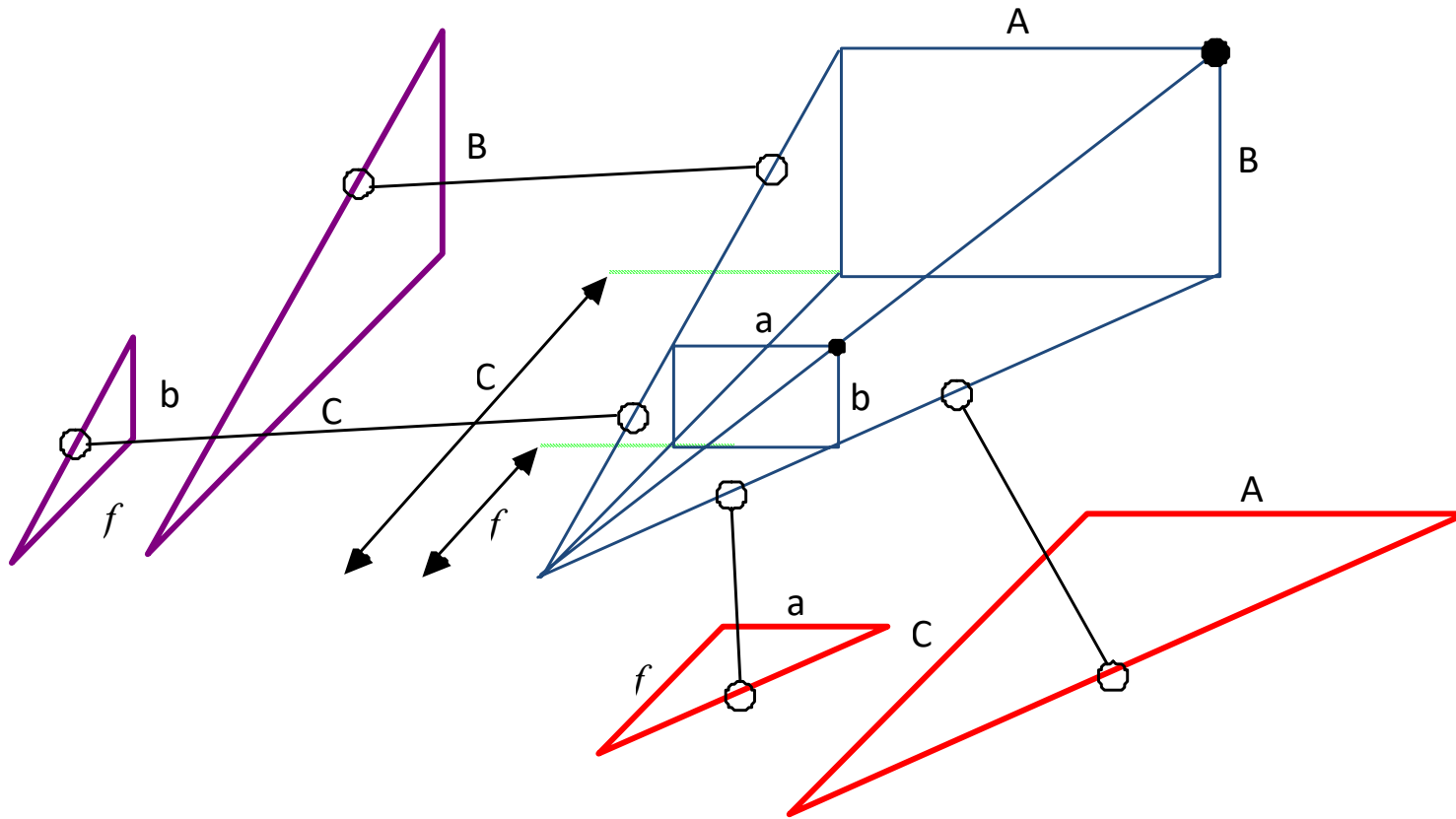
$$\frac{E}{F} = \frac{e}{f}$$

$$\frac{F}{D} = \frac{f}{d}$$

etc

# Solving Perspective Projection

- Similar triangles solves the relationship between 3-D space and 2-D image coordinates.
- Redraw the geometry once more, this time making apparent two pairs of **similar triangles**:



- By the **Similar Triangles Theorem**, we conclude that:

$$\frac{a}{f} = \frac{A}{C} \quad \text{and} \quad \frac{b}{f} = \frac{B}{C}$$

- OR:  $(a, b) = \frac{f}{C} \cdot (A, B) = (f A/C, f B/C)$

# Perspective Projection Equation

- The relationship between a 3-D point  $(X, Y, Z)$  and its 2-D image  $(x, y)$  :

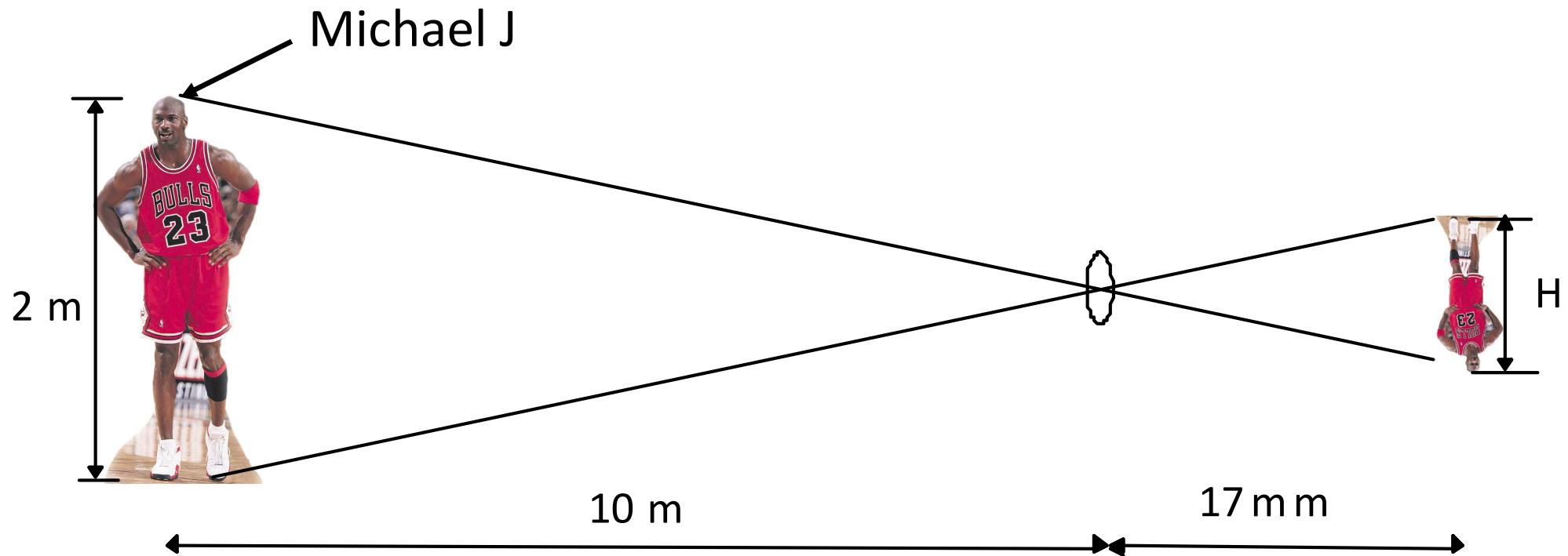
$$(x, y) = \frac{f}{Z} \cdot (X, Y)$$

where  $f$  = focal length

- The ratio  $f/Z$  is the **magnification factor**, which varies with the **range**  $Z$  from the lens center to the **object plane**.

# Perspective Projection Equation

- **Example**
  - A man stands 10 m in front of you.
  - He is 2 m tall.
  - Your eye's focal length is about 17 mm
- **Question:** What is the height  $H$  of his image on your retina?

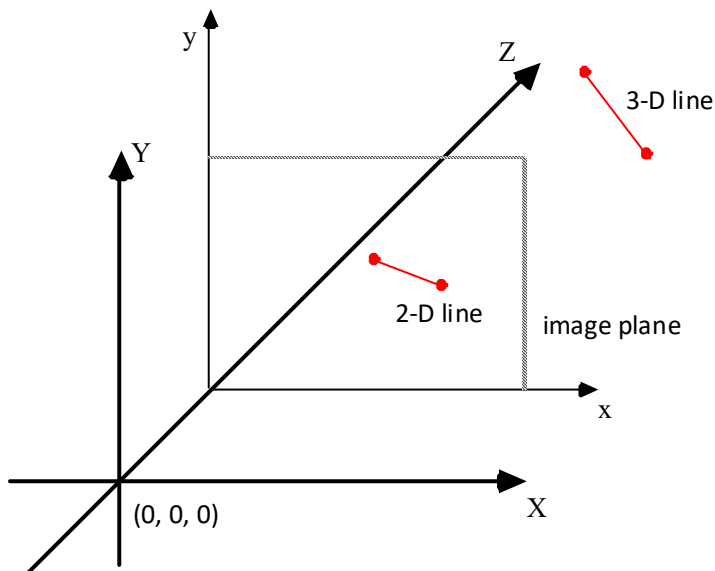


- By similar triangles,

$$\frac{2 \text{ m}}{10 \text{ m}} = \frac{H}{17 \text{ mm}} \quad \Rightarrow \quad \underline{H = 3.4 \text{ mm}}$$

# Straight Lines Under Perspective Projection

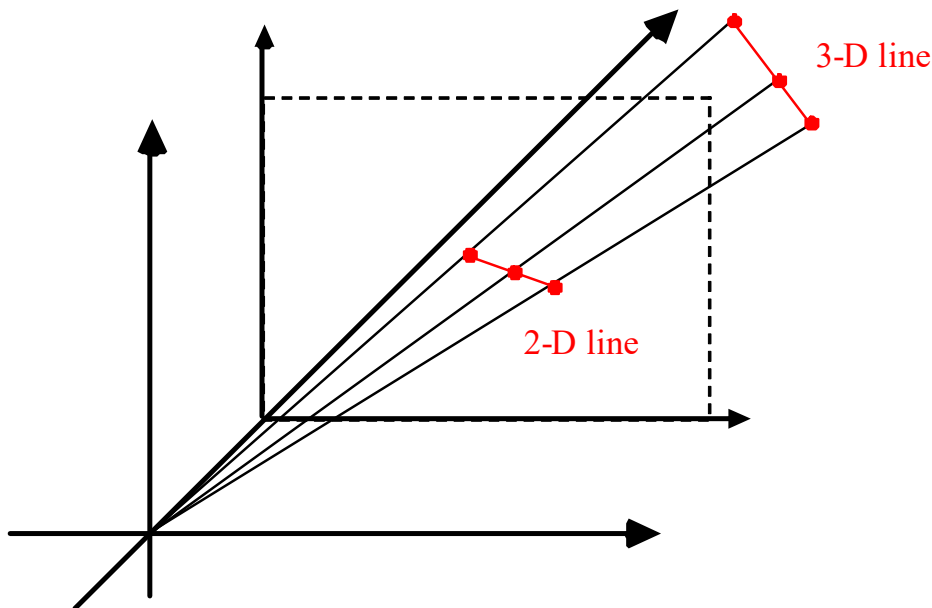
- Why do straight lines (or line segments) in 3-D project to straight lines in 2-D images?
- Not true of all lenses, e.g. "fish-eye" lenses do not obey the pinhole approximation.



To show this to be true, one could write the equation for a line in 3-D, and then project it to the equation of a 2-D line...

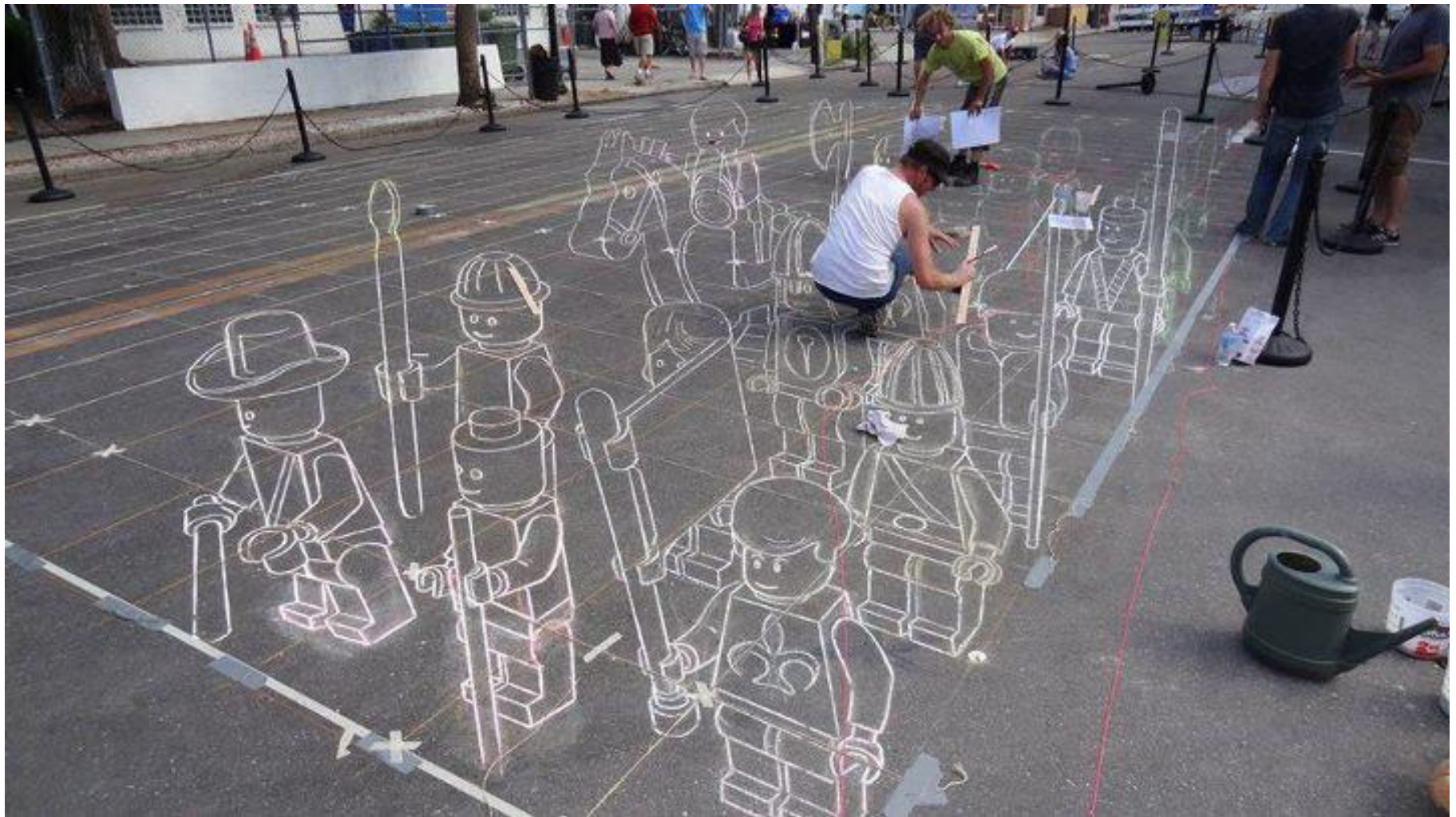
- Easier way:

- **Any two lines** touching the lens center and the 3-D line must lie in **the same plane** (a point and a line define a plane).
- The intersection of this plane with the image plane gives the projection of the line.
- The intersection of two (nonparallel) planes is a line.
- So, the projection of a 3-D line is a 2-D line.



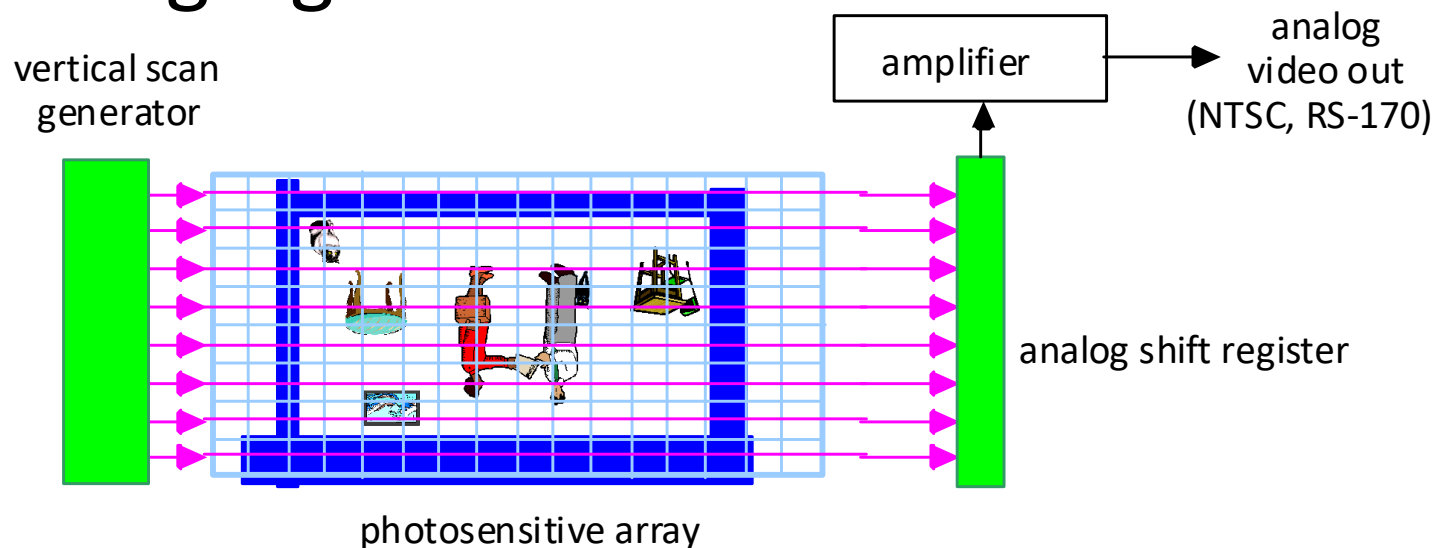
- In **image analysis**, this property makes finding straight lines much easier.

# Illusion of Perspective



# CCD Image Sensing

- Modern digital cameras sense **2-D images** using charge-coupled device (CCD) sensor arrays.
- The output is typically a line-by-line (raster) analog signal:

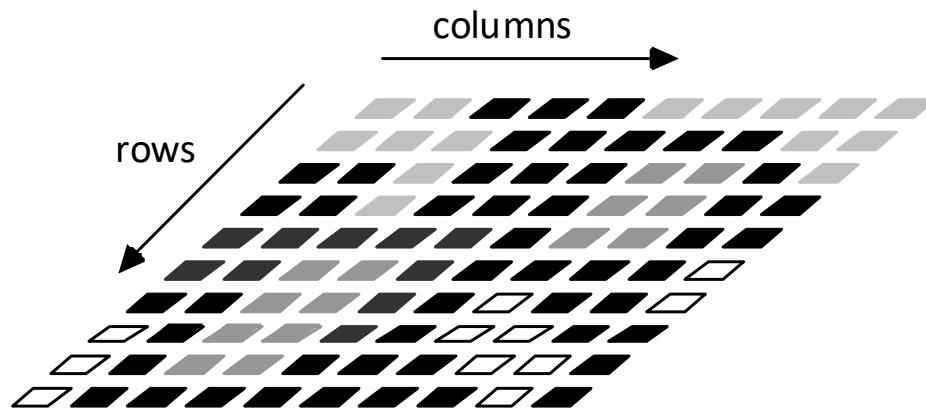


# A/D Conversion

- Consists of **sampling** and **quantization**.
- **Sampling** is the process of creating a signal that is defined only at **discrete points**, from one that is continuously defined.
- **Quantization** is the process of converting each sample into a **finite** digital representation.
- We will explore these in depth **later**.

# Digital Image

- A **digital image** is an array of numbers (row, column) representing image intensities



depiction of 10 x 10 image array

- Each of these **picture elements** is called a **pixel**.

# Sampled Image

- The image array is rectangular ( $N \times M$ )
- Examples: square images

128 x 128      ( $2^{14} \approx 16,000$  pixels)

256 x 256      ( $2^{16} \approx 65,500$  pixels)

512 x 512      ( $2^{18} \approx 262,000$  pixels)

1024x1024      ( $2^{20} \approx 1,000,000$  pixels)

# Sampling Effects

- It is essential that the image be sampled **sufficiently densely**; else the image quality will be severely degraded.
- Can be expressed mathematically via the Sampling Theorem, but the effects are **visually obvious**.
- With sufficient samples, the image **appears continuous.....**

# Spatial Downsampling

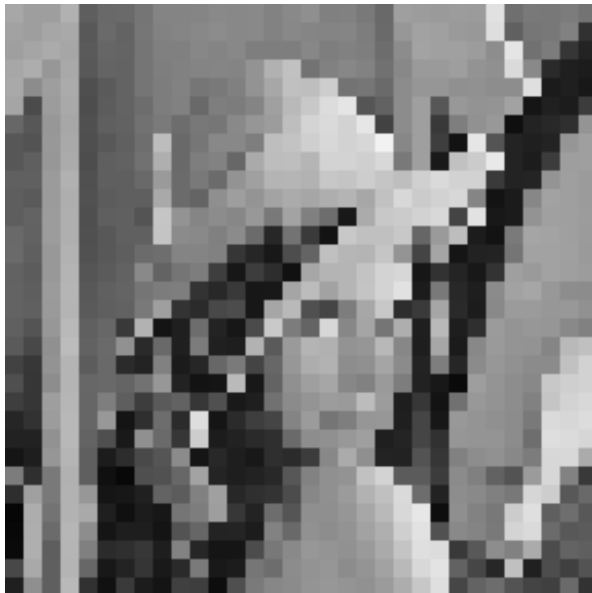
256 × 256



64 × 64



32 × 32



16 × 16

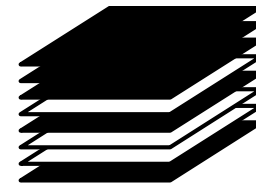


# Quantization

- Each **gray level** is quantized to an integer between 0 and  $K-1$ .
- There are  $K = 2^B$  possible gray levels.
- Each pixel is represented by  $B$  bits; usually  $1 \leq B \leq 8$ .



a pixel



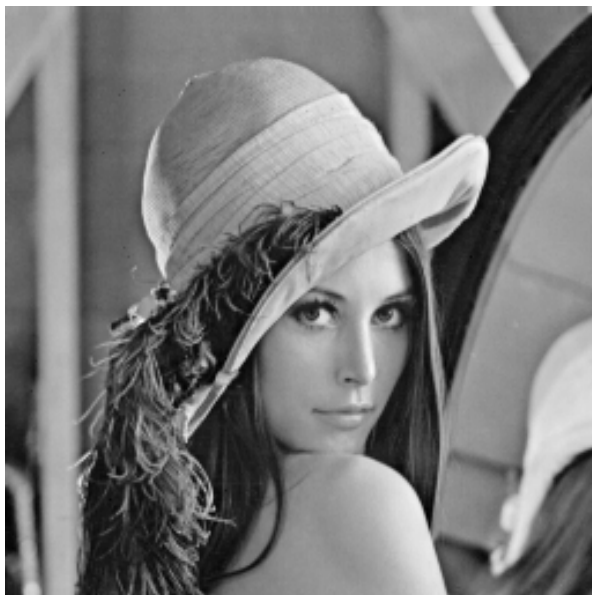
8-bit representation

# Quantization

- The pixel intensities or gray levels must be quantized **sufficiently densely** so that excessive information is not lost.
- This is **hard** to express mathematically, but again, quantization effects are **visually obvious**.

# Quantization

8 bits



4 bits



3 bits



2 bits



# The Image/Video Data Explosion

- Total **storage** required for **one digital image** with  $2^P \times 2^Q$  pixels spatial resolution and  $B$  bits / pixel gray-level resolution is  $B \times 2^{P+Q}$  bits.
- Usually  $B=8$  and often  $P=Q=10$  or more. Common image sizes may be  $> 10$  **megabyte**.

# The Image/Video Data Explosion

- Storing **1 second** of a 1024 x 1024 gray-level movie (TV rate = 30 images / sec) requires about 30 Mbytes.
- A 2-hour color theater-quality raw 4K digital video: (3 bytes/color pix) x (4096x2160 pix/frame) x (60 frames/sec) x (3600 sec/hr) x (2hr)  $\approx$  11.5 terabytes.
- Later, we will discuss ways to **compress** digital images and videos.

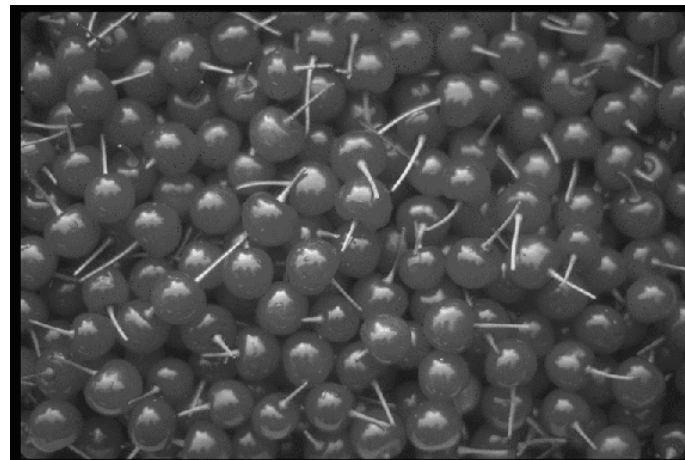
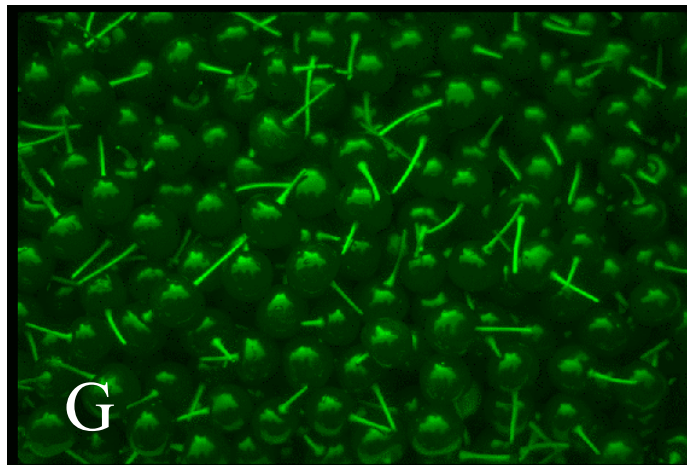
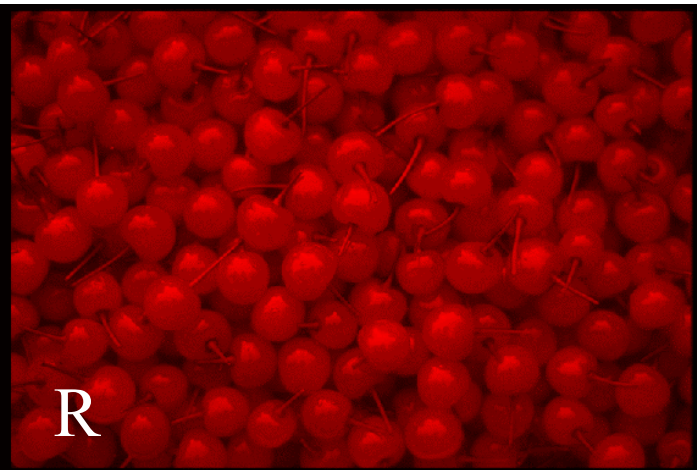
# Color Images

- A color image has vector-valued pixels. For example, 8 bits of Red, 8 bits of Blue, and 8 bits of Green.
- So you can think of a color image as a collection of three grayscale images.
  - ▶ E.g., a Red image, a Blue image, and a Green image.
  - ▶ These images are usually called color “bands” or “channels.”
  - ▶ Note: some color spaces like CMYK have *more* than 3 bands.
- In a grayscale image, the pixel values represent *intensity* or *luminance*, usually denoted I or Y.
- For an RGB color image, the intensity (luminance) is given by
$$I = 0.2989R + 0.5870G + 0.1140B$$
- This formula is not unique (i.e., it is not the only way to convert RGB to luminance), but it is **widely accepted**.
- Matlab provides a function `rgb2gray` to compute this.

# More on Color

- Some color spaces like YUV and YCbCr represent the grayscale image directly.
  - ▶ This was originally done so that black-and-white TV's could receive color TV signals... and just display the Y band.
- Many color image processing algorithms process the color bands independently like grayscale images.
  - ▶ For example, there may be noise that affects the bands independently.
  - ▶ This approach can produce weird changes in the colors, however.
- Often, it's desirable to modify an image without changing the colors.
  - ▶ For example, we might want to just make a color image brighter without making any visually obvious changes to the colors of the pixels.
  - ▶ In such cases, it's usually sufficient to apply the filter to the intensity image (grayscale image) only, then reconstruct the colors.
- We'll spend most of our time on grayscale images in this class.
- Development of true vector color filters is an active research area.

Color



Intensity

# Common Image Formats

- **JPEG (Joint Photographic Experts Group)** images are compressed with loss – see Module 7. All digital cameras today have the option to save images in JPEG format. File extension: *image.jpg*
- **TIFF (Tagged Image File Format)** images can be lossless (LZW compressed) or compressed with loss. Widely used in the printing industry and supported by many image processing programs. File extension: *image.tif*
- **GIF (Graphic Interchange Format)** an old but still-common format, limited to 256 colors. Lossless and lossy (LZW) formats. File extension: *image.gif*
- **PNG (Portable Network Graphics)** is the successor to GIF. Supports true color (16 million colors). File extension: *image.png*
- **BMP (bit mapped) format** is used internally by Microsoft Windows. Not compressed. Widely accepted. File extension: *image.bmp*
- **HEIC (High-Efficiency Image Container)** images are compressed using a single frame derivative of the HEVC video codec. File extension: *image.heic*.

# Neural Representations

- We may create many different types of **image representations**.
- These may rely on **features** like “edges,” “SIFT keypoints,” and so on (later!).
- Some of the most important are **neural representations** created by a **network**.
- **To study this**, we will start with ...

# Perceptrons

# The Perceptron

- The first “**neural network**” – invented by **Frank Rosenblatt**.
- It is a **binary classifier** (outputs ‘0’ or ‘1’) – i.e., it **makes decisions**.
- Decisions are computed in **two steps**:
  - A **linear combination** of input values
  - A **nonlinear thresholding** or ‘**activation**’ function

# Perceptron: Linear Step

- Given an **image**

$$\mathbf{I} = [I(i, j); 0 \leq i, j \leq N-1, M-1]$$

- Define any **subset**  $\mathbf{i} \subseteq \mathbf{I}$  which we will **vectorize**:

$$\mathbf{i} = \{i(p); 1 \leq p \leq P\}$$

- The **entire image**, or a **block**, or a **segmented region**, for example.

# Perceptron: Linear Step

- Define a set of **weights**

$$\mathbf{w} = \{w(p); 1 \leq p \leq P\}$$

then the **linear step** is simply (inner product)

$$\mathbf{w}^T \mathbf{i} = \sum_{p=1}^P w(p) i(p)$$

- We could have done this with **2D weights**  $\mathbf{W}$

$$\mathbf{W}^T \mathbf{I} = \sum_{m=0}^{M-1} \sum_{p=0}^{P-1} W(m,p) I(m,p)$$

# Perceptron: Nonlinear Step

- Given a **bias value**  $b$ , then the **nonlinear binary thresholding function**

$$f(\mathbf{i}) = \begin{cases} 1; & \mathbf{w}^T \mathbf{i} > b \\ -1; & \mathbf{w}^T \mathbf{i} < b \end{cases} = \text{sign}(\mathbf{w}^T \mathbf{i} - b)$$

is an example of an **activation function**. The activation function  $\text{sign}(\cdot)$  is called the **signum function**.

- A model of a **single neuron** only **activated** by a strong enough input.

# Perceptron: Training

- **Goal:** Find the weights to make the most accurate decisions.
- Need a set of  $Q$  **training sample images** with **labels** or **targets**:

$$\mathbf{T} = \left\{ (\mathbf{i}_1, t_1), (\mathbf{i}_2, t_2), \dots, (\mathbf{i}_Q, t_Q) \right\}$$

- The **training images**  $\mathbf{i}_q$  are labeled with **correct decisions/labels**  $t_q \in \{-1, 1\}$ .
- Given **weights**  $\mathbf{w}$  there is no **training error iff**

$$t_q \left( \mathbf{w}^T \mathbf{i}_q - b \right) > 0$$

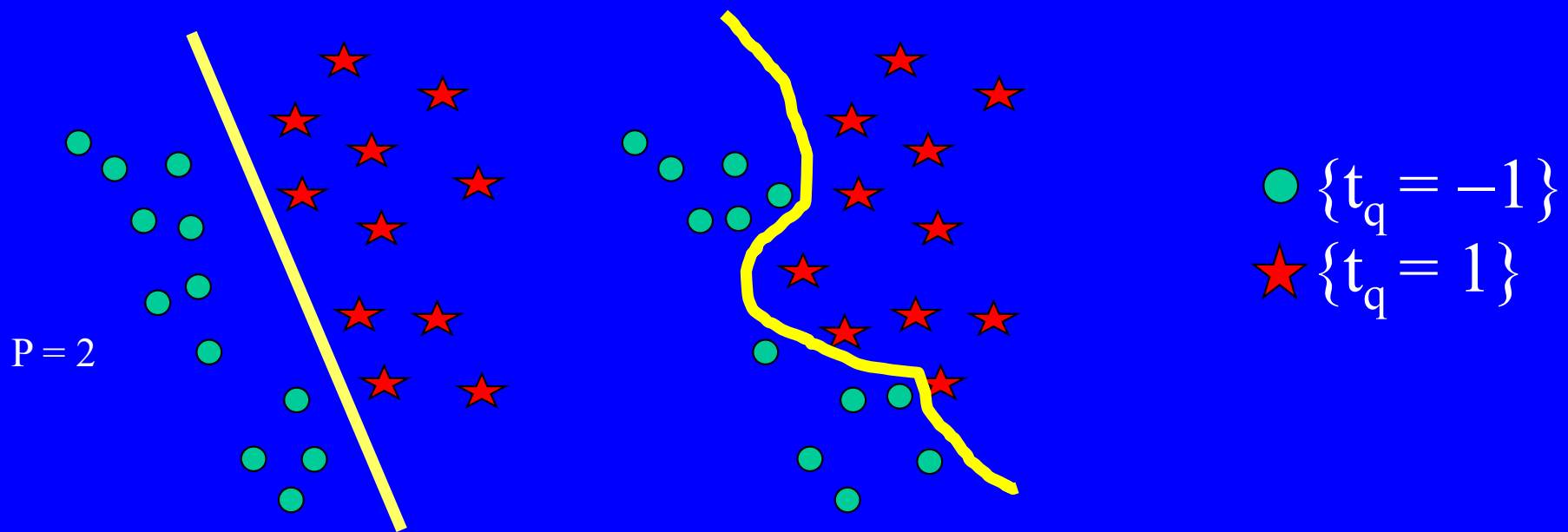
for all  $q = \{1, 2, \dots, Q\}$ .

# Perceptron: Optimization

- **Goal:** Find **weights** that minimize the **loss function**

$$L(\mathbf{w}, \mathbf{b}) = - \sum_{q=1}^Q t_q (\mathbf{w}^T \mathbf{i}_q - b)$$

- Turns out this is solvable only if the training set is **linearly separable:**



“Linearly” separable in P-space by a hyperplane of dimension (P-1)

Linearly Inseparable

# Gradient of Cost

- The **gradient** is the vector of **partial derivatives**

$$\nabla L(\mathbf{w}, \mathbf{b}) = \left( \frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \dots, \frac{\partial L}{\partial w_p}, \frac{\partial L}{\partial \mathbf{b}} \right)^T$$

- Basic idea: **optimize** the **perceptron** by **iteratively** following the **gradient down** to a (possible) **minimum**.

# Gradient Descent

- **Gradient descent:** A **simple** algorithm that iteratively moves current solution in the **direction** of the **gradient**:

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} - \gamma \nabla L(\mathbf{w}^{(n)}, \mathbf{b})$$

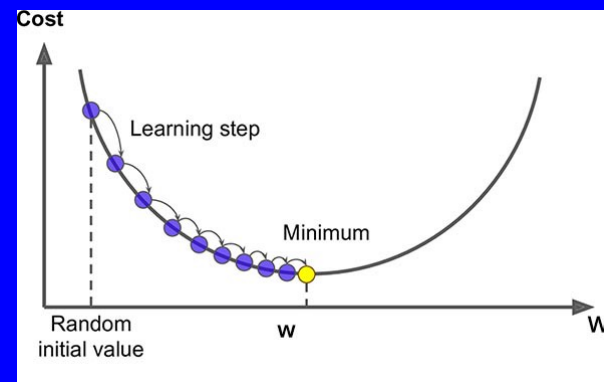
$$\mathbf{b}^{(n+1)} = \mathbf{b}^{(n)} - \gamma \frac{\partial L(\mathbf{w}^{(n)}, \mathbf{b}^{(n)})}{\partial \mathbf{b}}$$

or (exercise!)

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + \gamma t_q \mathbf{i}_q \quad q = 1, 2, \dots, Q$$

$$\mathbf{b}^{(n+1)} = \mathbf{b}^{(n)} + \gamma t_q$$

given an **initial guess** (say  $\mathbf{w}^{(0)} = 0$  or random).



Picture assumes a concave loss function.

- Here  $\gamma > 0$  is the **learning rate**. If too big it might not converge; too small, might converge very slowly). 73

# Problems with Perceptron

- First, there is **no lower** bound to the loss function! Makes numerical solution hard.
- **Rosenblatt's Solution**: Only iterate on those training images whose solution condition is violated at each step:

$$\mathbf{w}^{(n+1)} = \mathbf{w}^{(n)} + \gamma t_{q'} \mathbf{i}_{q'}$$

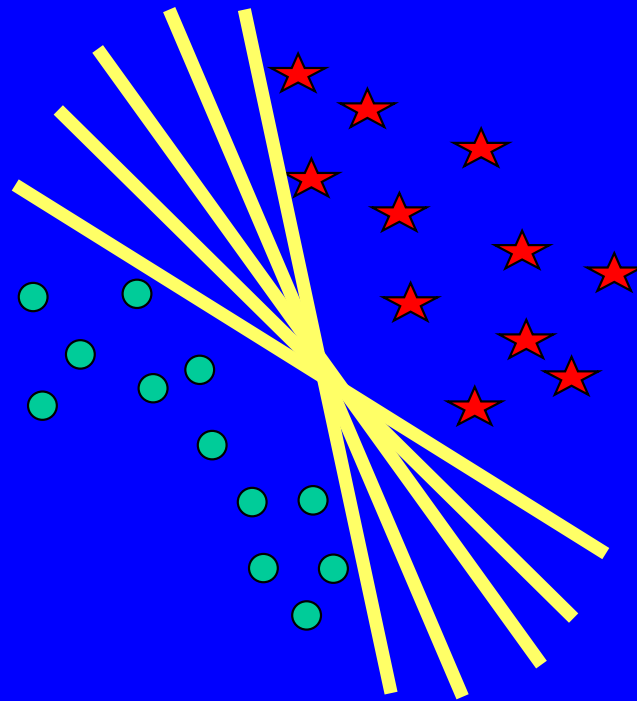
$$b^{(n+1)} = b^{(n)} + \gamma t_{q'}$$

for those  $q' = 1, 2, \dots, Q$  such that  $t_{q'} (\mathbf{w}^T \mathbf{i}_{q'} - b) < 0$ .

- **Does converge** but **only if the training set is linearly separable**. Which is rare, especially on **hard problems!**

# Can Converge Anywhere in “the Margin”

- The solution converged to is **not necessarily unique!**
- It can lie on **any line** (hyperplane) separating the **linearly separable classes**.



# Training Iteration

- Iterate by **repeatedly inputting** every image (set) in the training set (one “**epoch**”). Update **weights  $\mathbf{w}$**  on each training sample.
- **Standard Gradient Descent:** Iterate through  $\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_Q$  in indexed order each **epoch**.
- **Stochastic Gradient Descent:** Randomly re-order the training set of images  $\{\mathbf{i}_q\}$  after every epoch.
- Iterate a **fixed number of epochs**, or alternately until an **error**

$$\left\| \mathbf{f}_q^{(n)} - t_q \right\| \quad \mathbf{f}^{(n)}(\mathbf{i}_q) = \text{sign}\left(\mathbf{w}^{(n)\top} \mathbf{i}_q - \mathbf{b}^{(n)}\right)$$

is **small enough** (for some norm such as MSE).

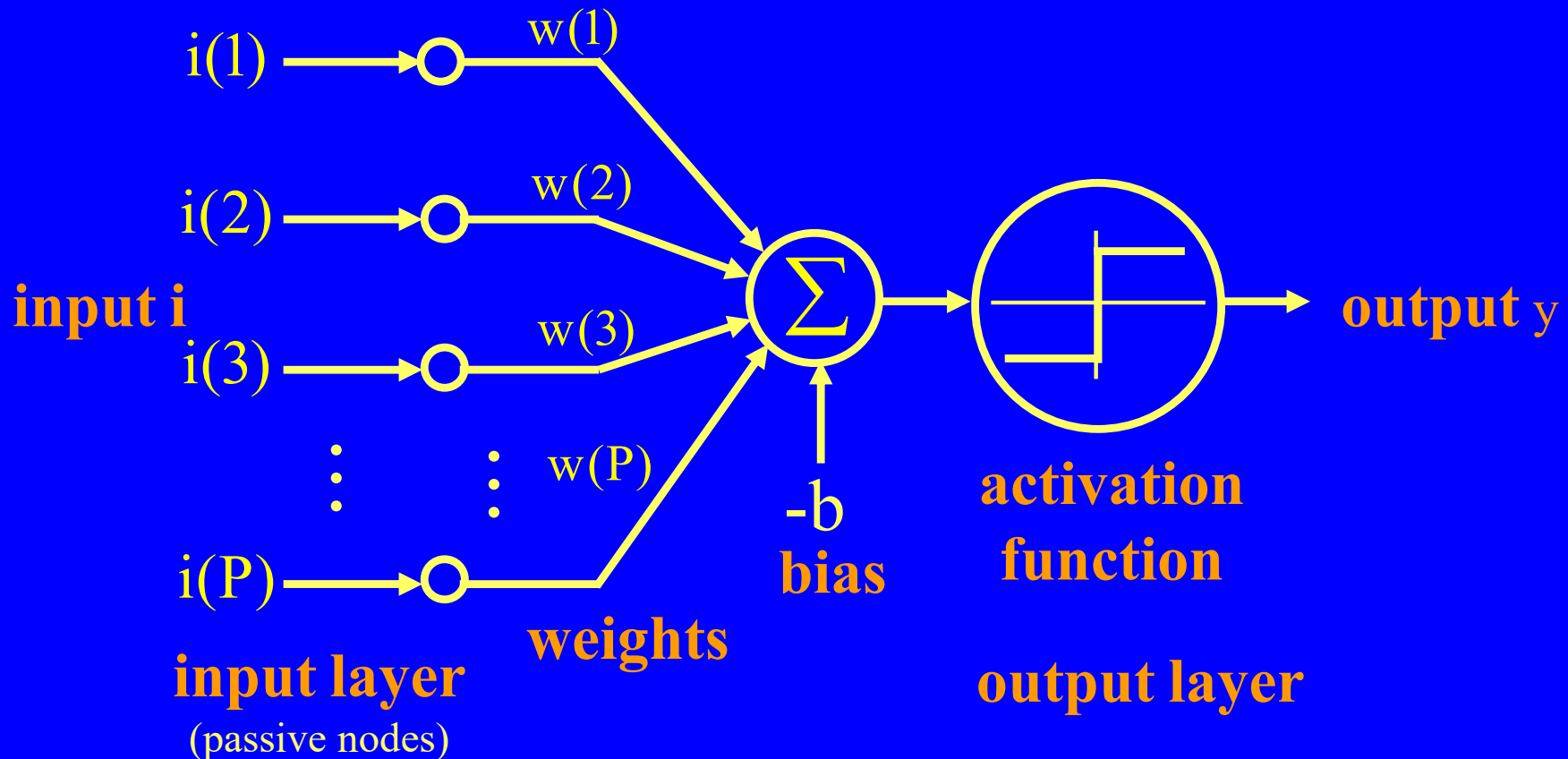
# Perceptron Diagram

- Given a (vectorized) **image** or **image piece**:

$$\mathbf{i} = \{i(p); 1 \leq p \leq P\}$$

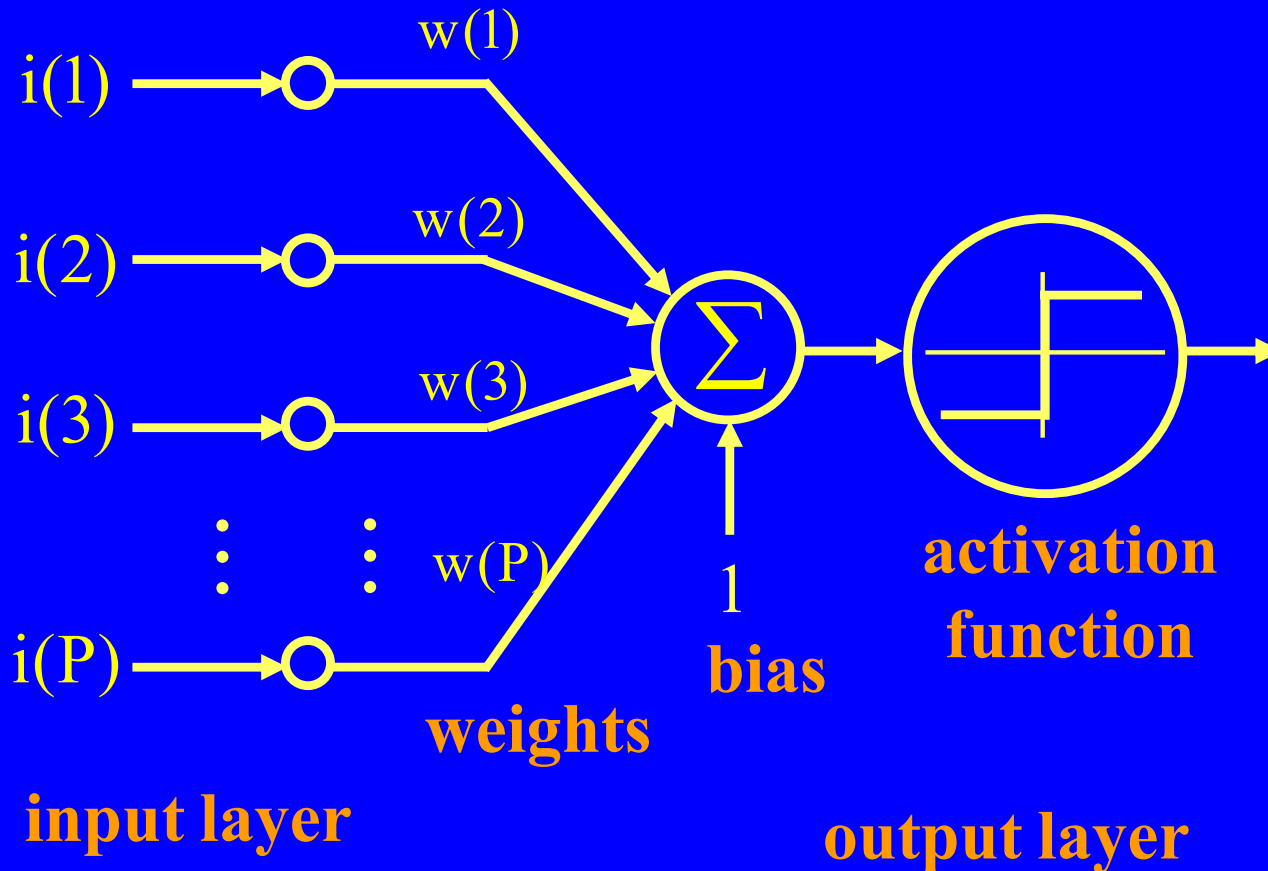
and a **trained** set of **weights**  $\mathbf{w}$  and **bias**  $b$ ,

$$\mathbf{w} = \{w(p); 1 \leq p \leq P\}$$



# Perceptron Diagram

- Convenient to **redraw** this way (notice the modified unit bias)

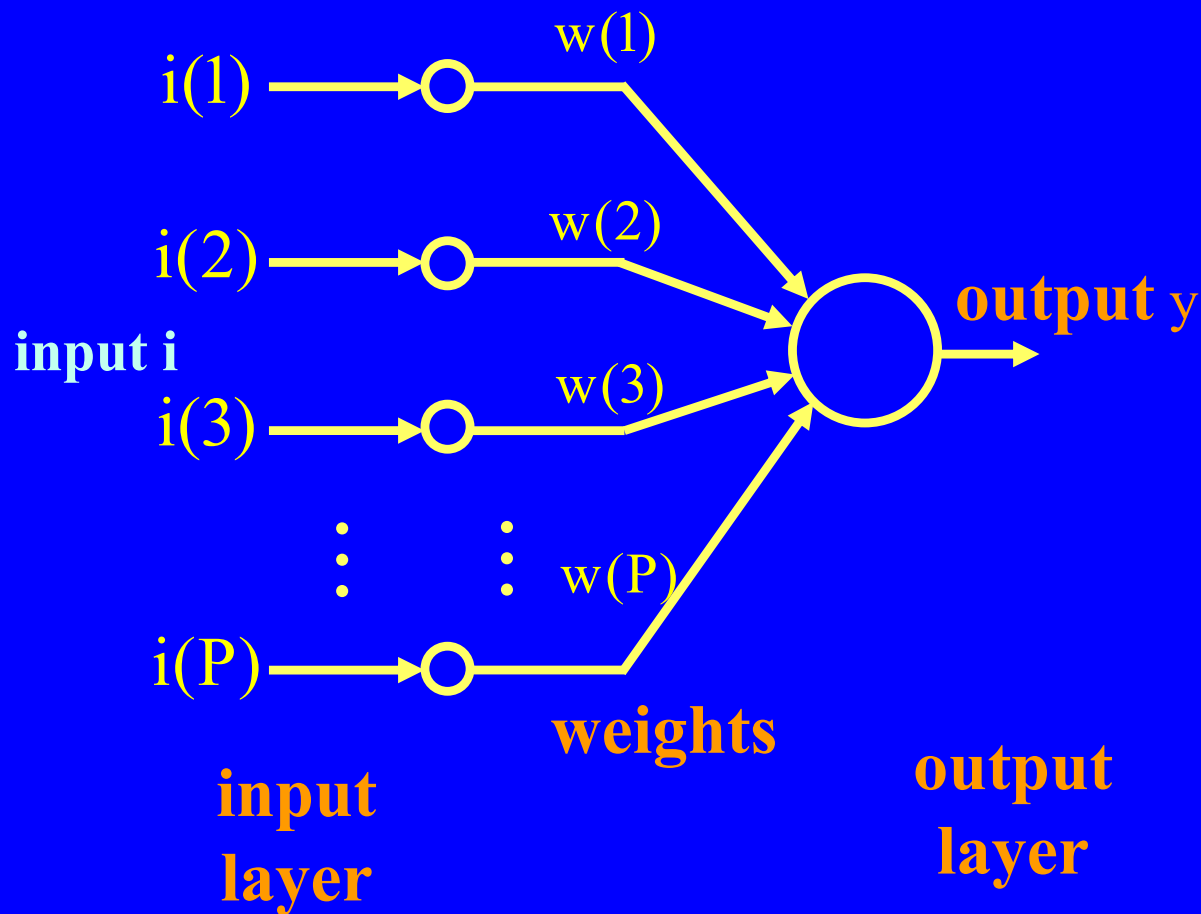


- **Output layer: sum and activation function**
- **Very common notation for general neural networks**

- The  $-\mathbf{b}$  can be **absorbed** into the other weights by **normalization** (more later).
- Sometimes the bias is drawn as another **(unit) input**.

# Simplified Perceptron Diagram

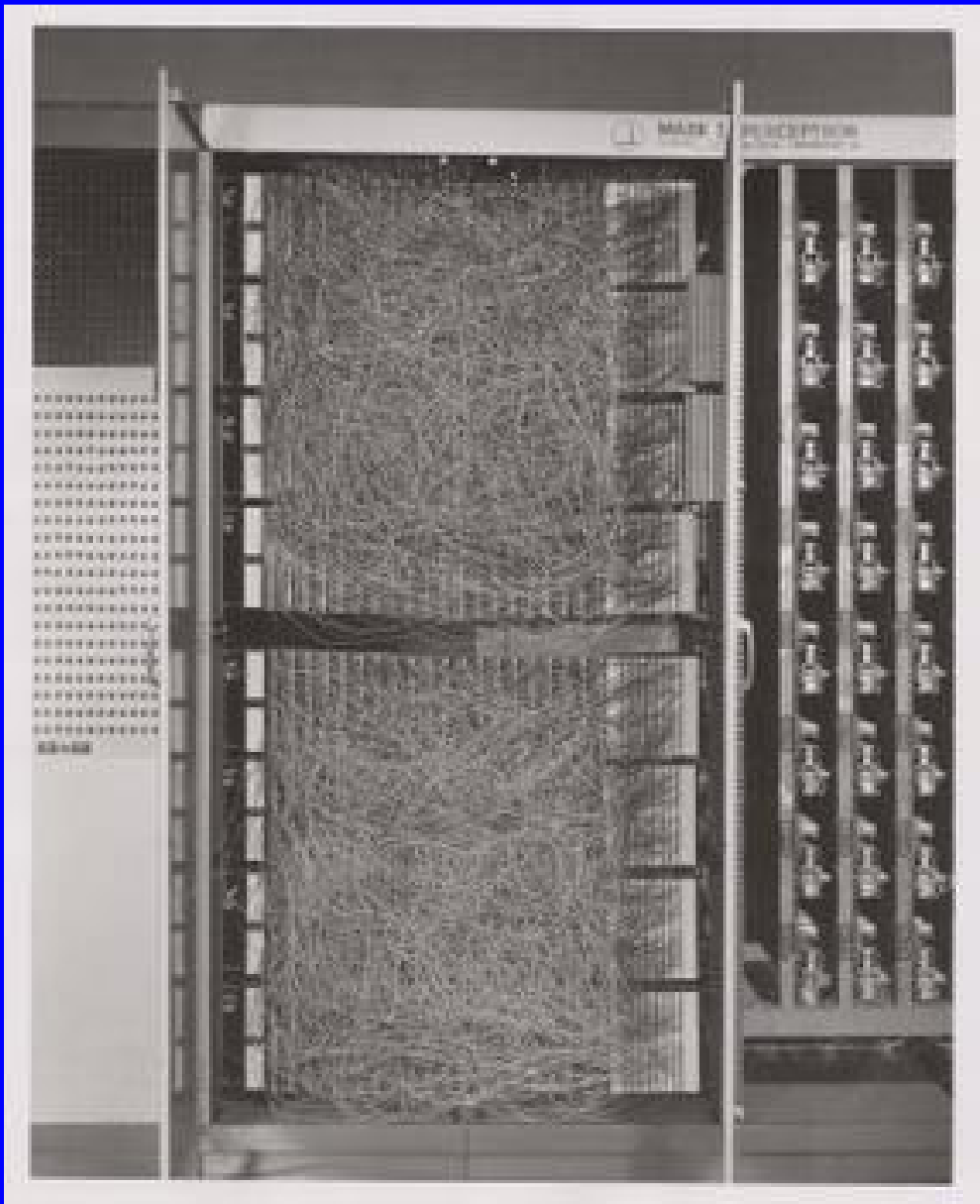
- If the **activation function is understood** (and we will be considering other ones!), can just draw this:



- **Input layer includes a constant bias**
- **Output layer: sum and activation function**
- **Very common notation for “vanilla” neural networks**

# Image Data

- **Later** we will **feed pixels** into learners, so we have been expressing **perceptron processing** in this way.
- However: perceptrons were usually not used for **image processing!** There weren't even **many digital images**.
- The **required computation** was **extremely formidable** for existing computers.
- It was tried: Rosenblatt and colleagues trained the “**Mark 1 Perceptron**” to recognize **20x20 digital images** using an IBM mainframe computer.
- It **could not recognize many patterns**, however.



# Promise of Perceptrons?

In 1958 pioneer **Frank Rosenblatt** said: “the perceptron is the embryo of an electronic computer that will .... be able to **walk, talk, see, write, reproduce itself, and be conscious of its existence.**”



- There were/are **significant problems** with perceptrons.
- But today, it is starting to look like he may have been **right!** (at least on some of it!)

# Failure of Perceptrons

- In the 1960s it was found that **perceptrons did not have much ability** to represent **even simple functions or data patterns.**
- **Marvin Minsky** and **Seymour Papert** showed it could not even represent the XOR function.
- Interest languished for a decade or more, hence **thus ended** the “**first wave**” of neural networks.
- Things changed when researchers began to **layer the networks.**

# Comments on Perceptrons

- We will discuss concepts like **training, validation, and testing** as we study more complex networks.
- **Perceptrons** were the **first neural networks** and the **first feed-forward neural networks** (the only kind we'll study here).
- They are **the basis** of today's modern **deep convolutional networks** (“ConvNets” or CNNs).

# Comments

- With this broad overview of topics related to image processing in hand, and a start on neural networks, we can proceed ...  
**onward** to **Module 2**.....