# Image Representation and Processing

# What's An Image?

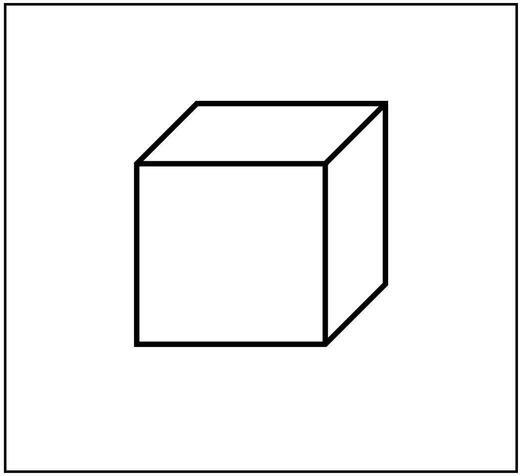


Image: distribution of light energy on 2D "film" : $E(x, y, \lambda, t)$

$(x, y)$ - position

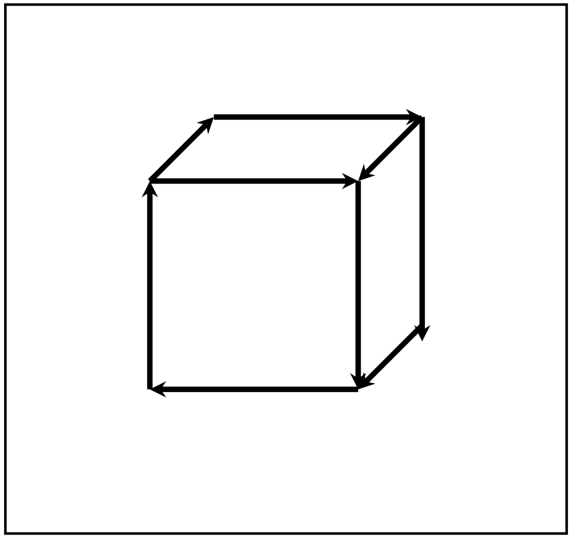
$\lambda$ - wavelength (blue, green, yellow, red, violet)

$t$ - time

This is a *continuous* representation

- Not easily represented on a computer

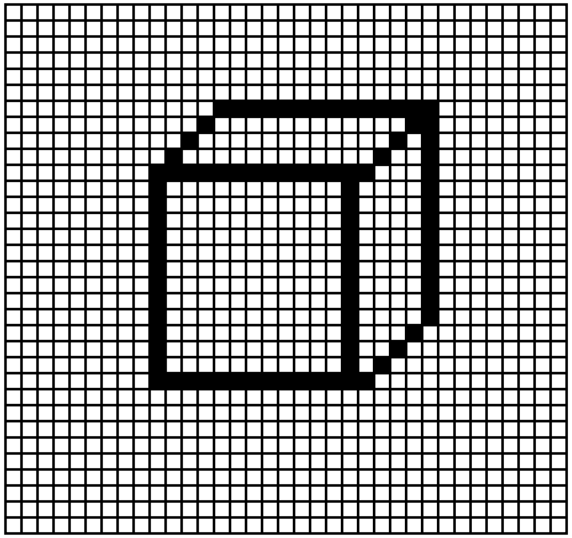
# *How Do We Represent Images?*



**Vector** Representation

+ arbitrary resolution
+ good for line drawings (text)

- may draw same point twice
- hard to do color changes



**Raster** Representation

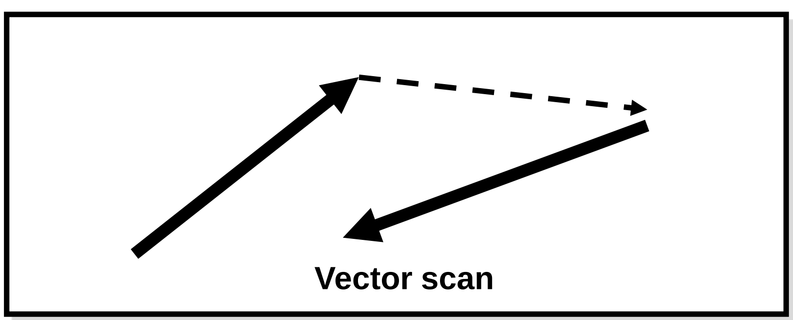
+ good for color images
+ general purpose

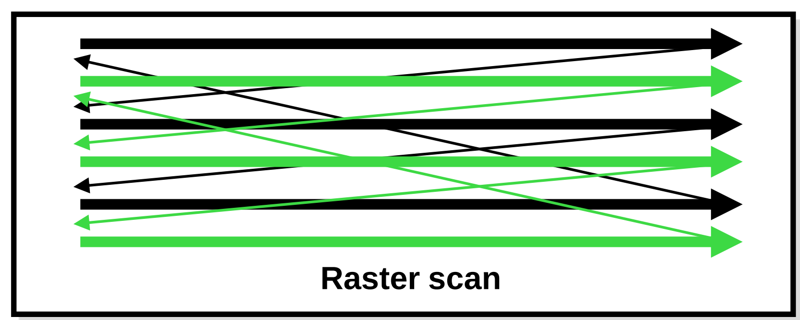
- bounded resolution (aliasing)
- store EVERY pixel

# *Vector and Raster*

- Early displays were *vector* displays
  - electron beam traces out line segments
  - image is a sequence of endpoints

- Raster displays (TV’s, LCD’s)
  - electron beam traces out a regular pattern: *raster s*
  - other raster technologies: LCD, plasma, micro-mirror
  - image is a raster: a 2D array of pixels

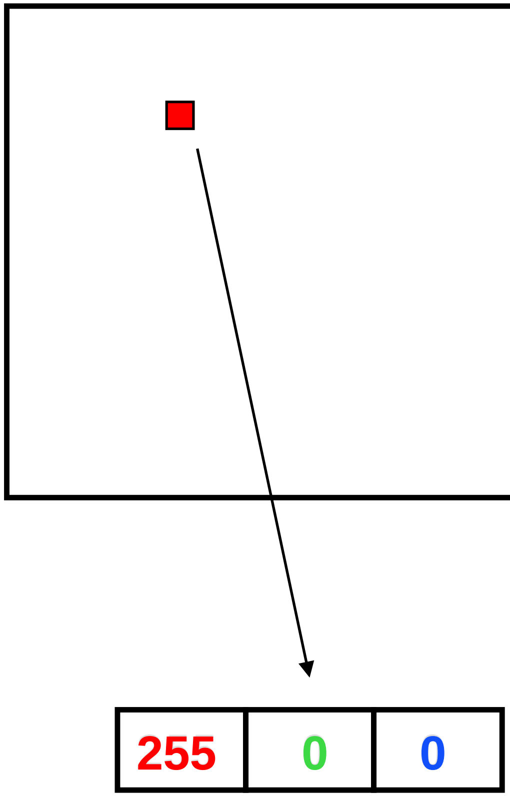


# Displays and Framebuffers

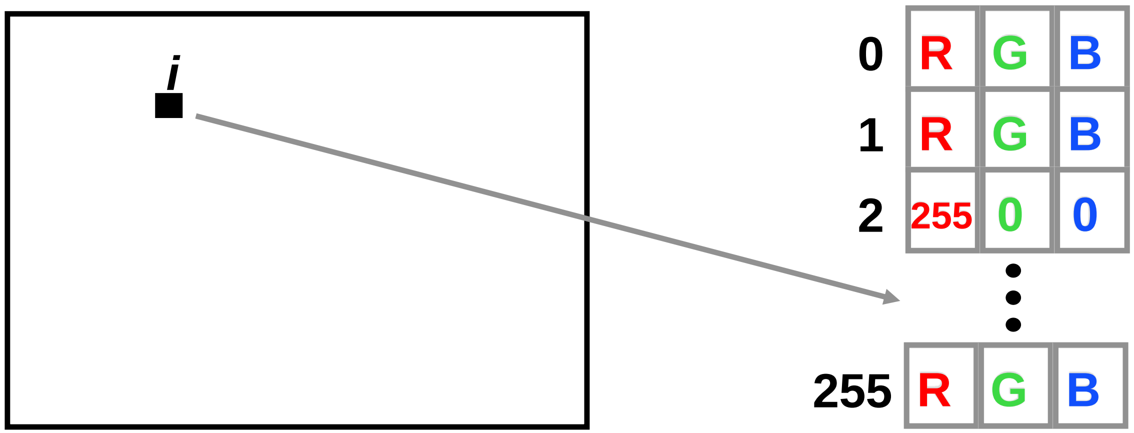
- The picture drawn by a raster display is stored in memory as a 2-D array of *pixels*.
- The value stored in each pixel controls the brightness of the beam (or beams, for color displays) as it sweeps past the corresponding screen location.
- The memory that stores the 2-D array of pixel values is called a *framebuffer*.
- The video hardware scans the framebuffer at ~60Hz
  - changes appear immediately
- Displays support different types of pixels
  - B&W displays: 1 bit/pixel *(bitmap)*.
  - Basic color displays: 8, 16, or 24 bits.
  - High-end displays: 96 or more bits.

# *Full-color (RGB) displays*




- For 24 bit color:
  - store 8 bits each of red, green, and blue per pixel.
  - E.g. (255,0,0) is pure red, and (255, 255, 255) is white.
  - Yields 2^24 = 16 million colors.
- For 15 bit color:
  - 5 bits red + 5 green + 5 blue
- The video hardware uses the values to drive the R, G, and B guns.
- You can mix different levels of R, G, and B to get (almost) any color you want

# Colormaps (LUT's)




- A single number (e.g. 8 bits) stored at each pixel.
- Used as an *index* into an array of RGB triples.
- With 8 bits per pixel, you get 256 colors of your choice
- Simple things to fill up color-maps with:
  - A gray ramp (for grayscale pictures)
  - A bunch of pre-chosen colors
  - A set of colors adaptively chosen for a given picture

# Some Picture File Formats

**JPEG**: Joint Photographic Experts Group Format

**TIFF**: Tagged-Image File Format

**GIF**: CompuServe Graphics Interchange Format

**PPM**: Portable PixMap Format (ASCII or binary)

**EPS**: Encapsulated PostScript Format (ASCII)

| | BITS PER PIXEL | FILE SIZE | COMMENTS |
|---|---|---|---|
| JPEG | 24 | small | lossy compression |
| TIFF | 8, 24 | medium | good general purpose |
| GIF | 1, 4, 8 | medium | popular, but 8-bit |
| PPM | 24 | big | easy to read/write |
| EPS | 1, 2, 4, 8, 24 | huge | good for printing |

Others: BMP, XPM, RAS, PICT, PNG, etc...

# Deeper Framebuffers

- Some frame buffers have 96 or more bits per pixel. What are they all for? We start with 24 bits for RGB.
- *Alpha channel*: an extra 8 bits per pixel, to represent "transparency." Used for digital compositing. That's 32 bits.

# Image Compositing

- Represent an image as layers that are composited (matted) together
- To support this, use pixel's extra *alpha* channel in addition to R, G, B
- Alpha is opacity: 0 if totally transparent, 1 if totally opaque
- Alpha is often stored as an 8 bit quantity; usually not displayed.
- Mathematically, to composite $a_2$ over $a_1$ according to matte $\alpha$

  $b = (1-\alpha) \cdot a_1 + \alpha \cdot a_2$

  $\alpha$ = 0 or 1 -- a hard matte, $\alpha$ = between 0 and 1 -- a soft matte
- Compositing is useful for photo retouching and special effects.
- Compositing is useful for translucent polygon rendering and volume rendering!

# Deeper Framebuffers

- Some frame buffers have 96 or more bits per pixel. What are they all for? We start with 24 bits for RGB.
- *Alpha channel*: an extra 8 bits per pixel, to represent "transparency." Used for digital compositing. That's 32 bits.
- A Z-buffer, used to hold a "depth" value for each pixel. Used for hidden surface 3-D drawing. 16 bits/pixel of "z" brings the total to 48 bits.
- Double buffering:
  - For clean-looking flicker-free real time animation.
  - Two full frame buffers (including alpha and z).
  - Only one at a time is visible—you can toggle instantly.
  - Draw into the "back buffer" (invisible), then swap.
  - Can be faked with off-screen bitmaps (slower.)
  - 2 x 48 = 96.

# *Image Processing*

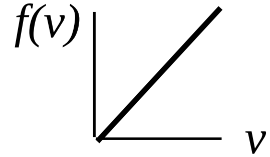
- *Point Processing*: modify each pixel as a function of its pixel value
- *Filtering*: output is a function of the (usually) local neighborhood around the pixel
- Image processing is a discrete version of signal processing (some lingo: image is a two-dimensional "signal")
- Other topics:
  - Image transformation (resize, warp)
  - Image compression
  - Texture mapping
  - …

# Point Processing

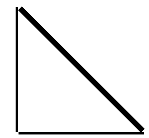
- Input: a[x, y], Output b[x, y] = f(a[x, y])
- f transforms each pixel value separately
- Useful for contrast adjustment

Suppose our picture is grayscale (a.k.a. monochrome).

Let $v$ denote pixel value, suppose it's in the range [0, 1].

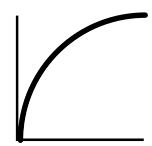



$f(v) = v$ **identity**; no change

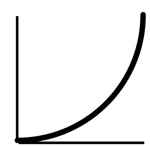


$f(v) = 1-v$ **negate** an image

(black to white, white to black)



$f(v) = v^p$, $p<1$ **brighten**



$f(v) = v^p$, $p>1$ **darken**

# Image Filtering: Blurring

original, 64x64 pixels

3x3 blur

5x5 blur

# Image Filtering:  Edge Detection

horizontal derivative

vertical derivative

# Image Filters

- In 1-D such a simple filter can be written:

$$b[x] = \sum_{t=-\infty}^{+\infty} a[t]h[x-t]$$

where $a[x]$ = input signal

$b[x]$ = output signal

$h[x]$ = filter

$x$ takes on only integer values

- This is convolution, written b = a⊗h for short. Convolution is commutative, i.e. a⊗h=h⊗a
- 2-D is similar, but with a double-summation:

$$b[x,y] = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} a[u,v]h[x-u, y-v]$$

- This class of filters is called "linear, shift-invariant"

# Image Filter Example

$$b[x,y] = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} a[u,v]\,h[x-u, y-v]$$

*a[x,y]* = input signal

*b[x,y]* = output signal

*h[x,y]* = 3x3 filter

*x,y* take on only integer vals

$\frac{1}{4}$ ×

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

**h**

*(0,0) at center*

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |
| 0 | 0 | 0 | 0 | 0 | 25 | 25 | 25 | 25 | 25 |

**a**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0? | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25? | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0? | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 25 | 25 | 0 | 0 | 0 | 0 |

**b**

# Blurring Filters

A simple blurring effect can be achieved with a 3x3 filter centered around a pixel,

written explicitly:

```
b[x, y] = (a[x-1, y-1] + a[x, y-1] + a[x+1, y-1]
        +a[x-1, y] + a[x, y] + a[x+1, y]
        +a[x-1, y+1] + a[x, y+1] + a[x+1, y+1])  / 9
```

or as coefficient matrix:

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

More blurring is achieved with a wider *n×n* filter:

$$\frac{1}{n*n}\begin{matrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{matrix}$$

# Edge Filter

To find edges, use approximation to the magnitude of the gradient of the image.

Gradient and its magnitude:

$$\nabla a = \frac{\partial a}{\partial x} \quad \frac{\partial a}{\partial y}, \qquad |\nabla a| = \sqrt{\frac{\partial a}{\partial x}^2 + \frac{\partial a}{\partial y}^2}$$

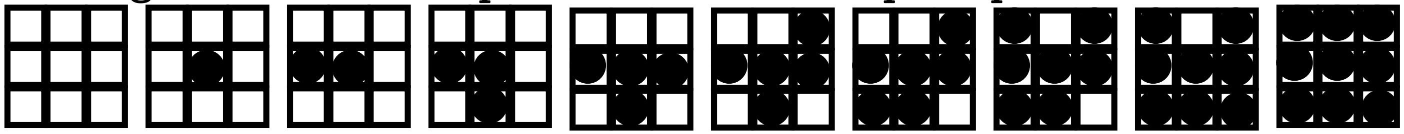
Sobel edge filter uses these weights:

$$\frac{\partial}{\partial x} \Rightarrow \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix}, \qquad \frac{\partial}{\partial y} \Rightarrow \begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

This is a *nonlinear filter* because of the sqrt and square operations.

# *Image Display and Print*

- How to draw grayscale on a 1-bit screen, or full color on an 8-bit screen
- Basic idea: give up *spatial* resolution in return for greater *brightness* or *color* resolution
- The eye does *spatial averaging*, so present a pattern whose *average* color matches the color you want
- In the patterns below, each square is either black or white.
  - From far away, the eye sees the average brightness of each grid, not the individual squares.
  - The average brightness of each 3x3 grid depends on the number of black and white squares—you can get ten distinct brightness levels ranging from black to white.
  - To draw a grayscale picture, each input pixel is represented by an ouput pattern. The pattern of dots that gets drawn depends on the input pixel value.



0 1 2 3 4 5 6 7 8 9

# Halftone Screens

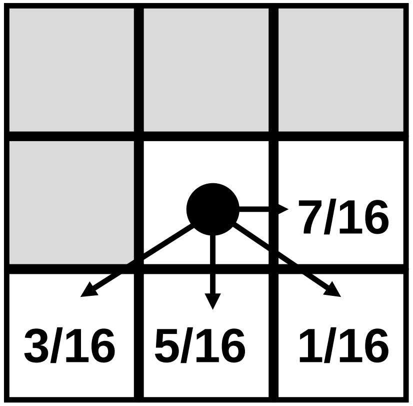
- How do we select a good set of patterns
- Pick patterns that avoid annoying artifacts:
  - Constant-brightness regions should not have obvious stripes.
  - On many devices (e.g. laser printers) isolated pixels should be avoided.
  - Growth-sequence: pixels that are "on" at one brightness levels should remain on for all higher levels. This avoids contouring artifacts.
- The full set of dot patterns can be encoded in a single n x n *dither matrix*. Each element in the matrix is a threshold: the dot is turned on for input values greater than the threshold. A sample 3x3 dither matrix is

| | | |
|---|---|---|
| 6 | 8 | 4 |
| 1 | 0 | 3 |
| 5 | 2 | 7 |

# Floyd-Steinberg Error Diffusion

If image and display have the same resolution:

- The values of the input image's pixels are normalized in floating point format to [0,1] with 0 (black) and 1 (white).
- Scan in raster order.
- At each pixel, draw the least-error output value (round off.)
- Divide the error into 4 (uneven) chunks.
- Add the error chunks back into the input values, at the 4 neighboring pixels you haven't hit yet:
- Can alternate scan direction



# *Floyd-Steinberg Error Diffusion*

Original image

After Floyd-Steinberg dithering

# Color Dithering

- You can mix Red, Green, and Blue to get any color you like.
- If you have an RGB image and a 3bit display, 1 per color, you just dither R, G, and B separately.
- On an 8 bit display, you can use the color map to divide the 8 bits into three parts (3, 3, 2) for R, G, and B. (Blue gets shortchanged because we can’t see blue very well.) So you get 8 levels each for R and G, and 4 for B.
- Dither R, G, and B separately (Floyd-Steinberg works fine for multi-bit output,) assemble the results into an 8-bit byte, and write to the frame buffer.
- The results generally look excellent, particularly on a high-res monitor.

More on dithering:

http://www.iro.umontreal.ca/~ostrom/publications/research.html#halftoning