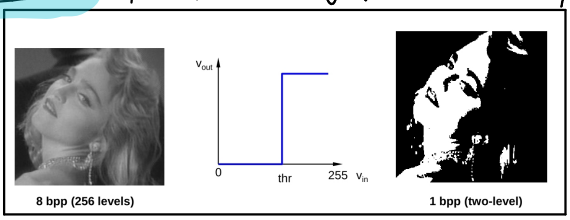
Slide 6: Digital Halftoning

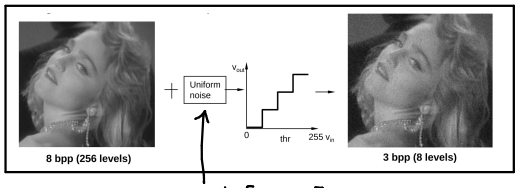
Thresholding: easy way to turn grayscale to binary

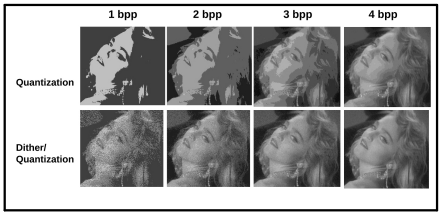


← loss of info

Unordered Dither

- add uniformly distributed white noise (dither signal) to input image by quantization
- dither hides artifacts





some random # $[-m, m]$

$$m = \frac{\text{MX GRAY LVLS}}{\text{quantization lvls}}$$

Ordered Dithering: Can simulate gray by some black/white dotting pattern!!!

Dither Matrix

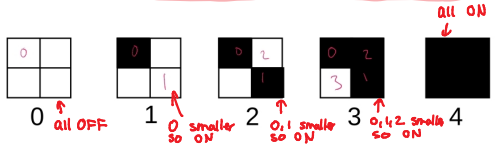
$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

From D^2 and D^3, can use this Eq to find larger Dithers:

$$D^{(n)} = \begin{bmatrix} 4D^{(n/2)} + D_{00}^{(2)} U^{(n/2)} & 4D^{(n/2)} + D_{01}^{(2)} U^{(n/2)} \\ 4D^{(n/2)} + D_{10}^{(2)} U^{(n/2)} & 4D^{(n/2)} + D_{11}^{(2)} U^{(n/2)} \end{bmatrix}$$

How to color Dither matrix

"!! The arrangement of these dots simulates gray!!!"



2x2 area creates 5 intensity lvls

nxn pixel area creates n^2+1 intensity lvls

① from 2x2 dither matrix

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

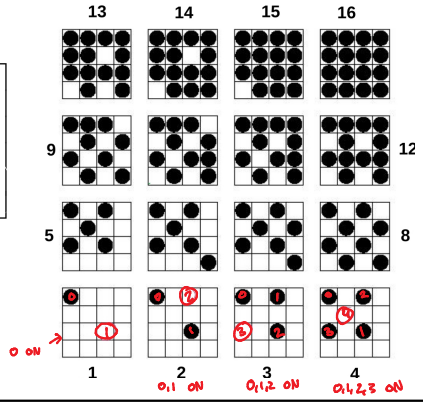
② Using Eq to find 4x4 or D^4 Dither Matrix

$4D^{(n/2)} + D_{00}^{(2)} U^{(n/2)} \rightarrow \begin{bmatrix} 0 & 8 \\ 12 & 4 \end{bmatrix} + 0$ $\quad 4D^{(n/2)} + D_{10}^{(2)} U^{(n/2)} \rightarrow \begin{bmatrix} 0 & 8 \\ 12 & 4 \end{bmatrix} + 3$

$4D^{(n/2)} + D_{01}^{(2)} U^{(n/2)} \rightarrow \begin{bmatrix} 0 & 8 \\ 12 & 4 \end{bmatrix} + 2$ $\quad 4D^{(n/2)} + D_{11}^{(2)} U^{(n/2)} \rightarrow \begin{bmatrix} 0 & 8 \\ 12 & 4 \end{bmatrix} + 1$

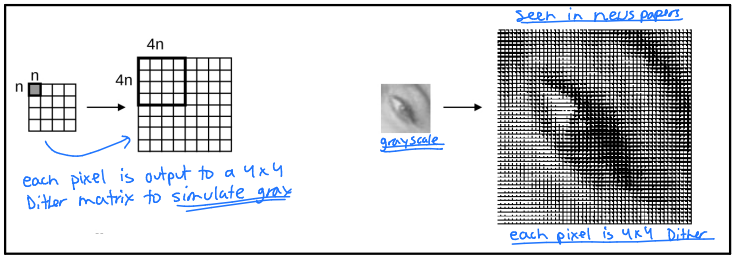
③ Make D^4

$$D^{(4)} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$



Patterning

- let output image > input img
1) quantize input $[0 \ldots n^2]$ gray lvls
2) Treshold each pixel against all entries in Dither matrix
   - each pixel → 4×4 block of black/white dots (D^4 matrix)
   - ∴ n×n input pixel → 4n × 4n output image
- If a large input area has same constant val → output same val



Implementation

- say input size = output size
1) Quantize input image to $[0 \ldots n^2]$ gray lvls
2) compare Dither matrix with input image

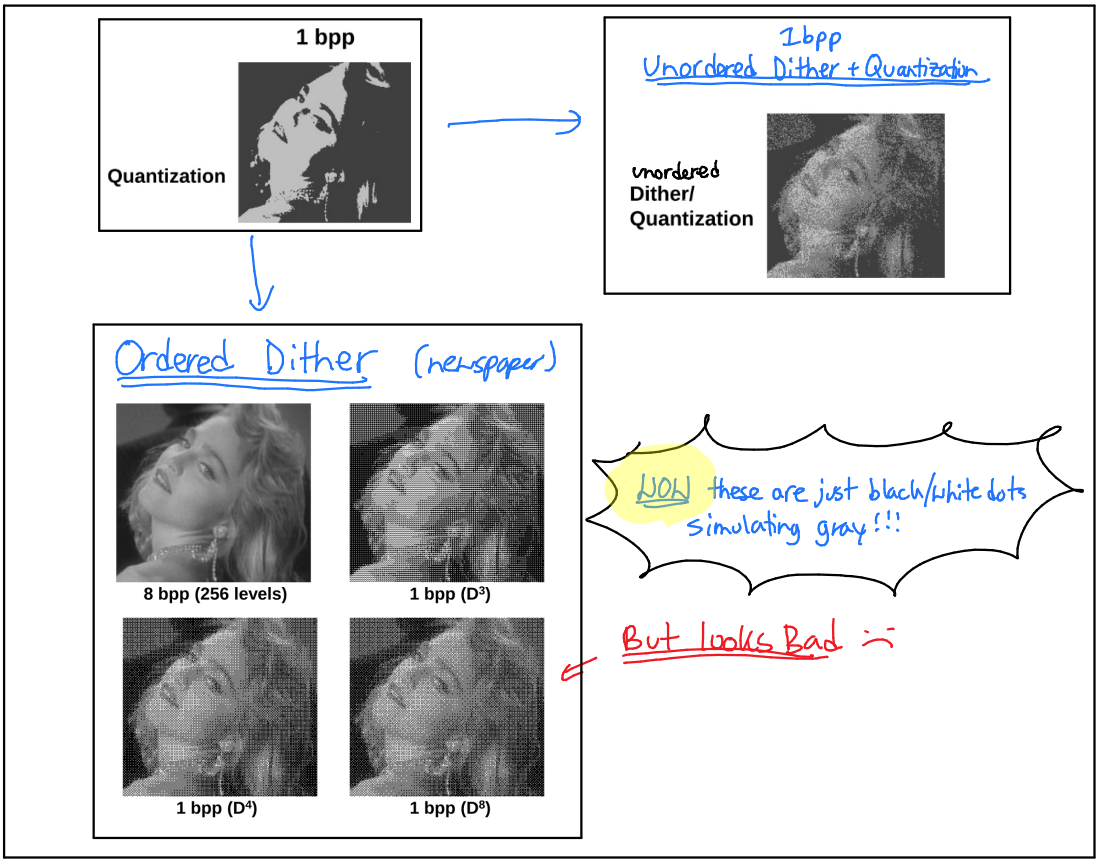
```
for(y=0; y<h; y++)        // visit all input rows
  for(x=0; x<w; x++){     // visit all input cols
      i = x % n;          // dither matrix index
      j = y % n;          // dither matrix index

      // threshold pixel using dither value D_ij^(n)
       out[y*w+x] = (in[y*w+x] > D_ij^(n))? 255 : 0;
}
```

h = row length ← Y = row
w = col length ← x = col

X loop: goes thru each elem in 1st row
Y loop: goes to next row

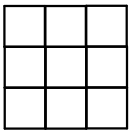
Y*w = go down each row
bla + x = go to each elem in row



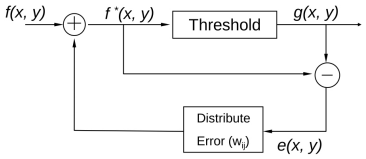
Error Diffusion : spreading error to neighbors

- Whenever we turn a grayscale image to just black or white → we get an error bc. crushing scales to 0 or 255
- To compensate of overshoots (255) and undershoots (0) → need to spread error to neighbors

ex. Can only make pixels 0 or 255
If input pxl is 130 → make it white (255)
- that excessive brightness ($\frac{255-30}{=125}$) is an error!
- 125 must be subtracted from neighbors!
- Why? since I made pixels brighter
  → make neighbors darker to compensate for excessive brightness



Floyd-Steinberg Algorithm



corrected pixel value + Sum up the grayscale errors at this pixel = "corrected intensity value"

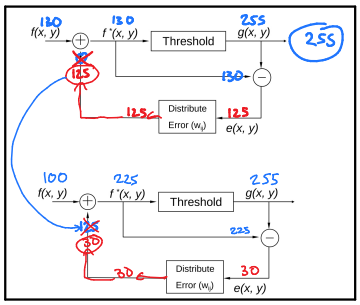
$$f^*(x,y) = f(x,y) + \left[\sum_i \sum_j w_{ij} e(x-i, y-j)\right] = \text{"corrected intensity value"}$$

making pixel 0 or 255 based on this

$$g(x,y) = \begin{cases} 255 & \text{if } f^*(x,y) > MXGRAY/2 \\ 0 & \text{otherwise} \end{cases}$$

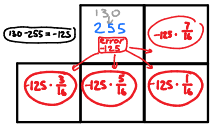
$$e(x,y) = f^*(x,y) - g(x,y)$$

$$\sum_i \sum_j w_{ij} = 1$$



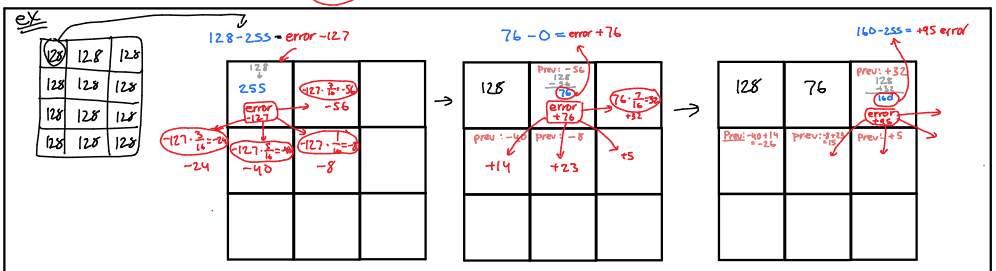
Floyd-Steinberg

Pass the errors to these boxes



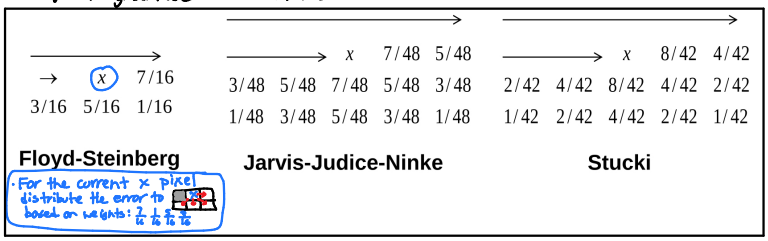
error diffusion
- each box gets some fraction of the error
- add $-125 \cdot \frac{7}{16}$ to the next box etc...

ex.



Error Diffusion Weights

- moves left to right
- sum of all diffused error weights = 1
- larger neighborhood → better results



Floyd-Steinberg | **Jarvis-Judice-Ninke** | **Stucki**

- For the current x pixel distribute the error to based on weights: $\frac{7}{16}$ $\frac{1}{16}$ $\frac{5}{16}$ $\frac{3}{16}$

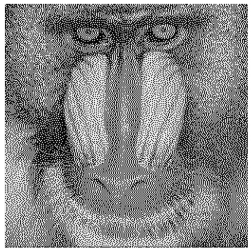
Examples (1)

Floyd-Steinberg

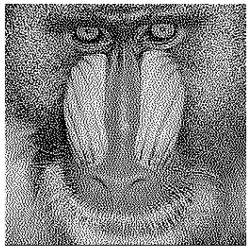
Jarvis-Judice-Ninke

Examples (2)

Floyd-Steinberg



Jarvis-Judice-Ninke



Implementation

```
thr = MXGRAY /2;                    // init threshold value
for(y=0; y<h; y++){                 // visit all input rows
 for(x=0; x<w; x++) {               // visit all input cols
    *out = (*in < thr)?             // threshold
           BLACK : WHITE;           // note: use LUT!

    e = *in - *out;                 // eval error
    in[ 1 ] +=(e*7/16.);            // add error to E  nbr
    in[w-1] +=(e*3/16.);            // add error to SW nbr
    in[ w ] +=(e*5/16.);            // add error to S  nbr
    in[w+1] +=(e*1/16.);            // add error to SE nbr

    in++;                           // advance input  ptr
    out++;                          // advance output ptr
 }
}
```

Border Issue Fix

Question: How do we deal with edges?

Issues: 1) errors can be deposited beyond image border (true for all neighborhood operations :-)

2) the depositing of errors can force greyvals to go out of [0,255] range.

Solutions: a) Before you deposit errors: do an IF statement:

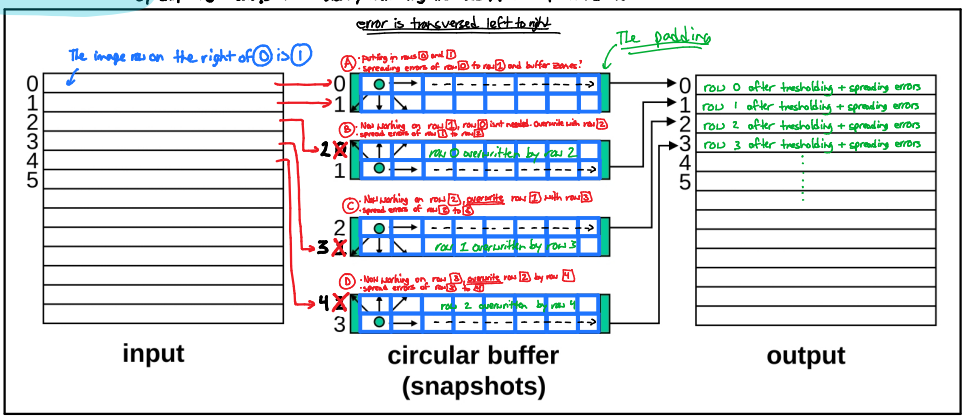
```
if (x < w-1 and y < h-1) {
    e = *in - *out;
    in[ 1 ] +=(e*7/16.);
    in[w-1] +=(e*3/16.);
    in[ w ] +=(e*5/16.);
    in[w+1] +=(e*1/16.);
}
```

b) find image with extra rows and cols

standards: • excessive memory
• need 16-bit precision (short) for pixel values outside [0,255]

Solution: use Circular buffer to significantly reduce memory req.

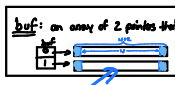
Circular Buffer: to fix padding excessive memory issue by: by only using 2 arrays and constantly overwriting the used pixel row with new row



median, convolve
use this
assume buffer not over 32

(better) Better Implementation Code

buf: an array of 2 pointers that point to 2 memory locations



```
thr = MXGRAY /2;                    // init threshold value
copyRowToCircBuffer(0);             // copy row 0 to circular buffer
for(y=0; y<h; y++){                 // visit all input rows
  copyRowToCircBuffer(y+1);         // copy next row to circ buffer
  in1 = buf[  y  %2] + 1;           // circ buffer ptr; skip over pad
  in2 = buf[(y+1)%2] + 1;           // circ buffer ptr; skip over pad
  for(x=0; x<w; x++) {              // visit all input cols
       *out = (*in1 < thr)? BLACK : WHITE;   // threshold

       e = *in1 - *out;             // eval error
       in1[ 1] +=(e*7/16.);         // add error to E  nbr
       in2[-1] +=(e*3/16.);         // add error to SW nbr
       in2[ 0] +=(e*5/16.);         // add error to S  nbr
       in2[ 1] +=(e*1/16.);         // add error to SE nbr

       in1++; in2++                 // advance circ buffer ptrs
       out++;                       // advance output ptr
  }
}
```

Copy row 0 to circular buffer

If I iterate by rows (no more rows and col)

1) copy next row to buffer
2) input pixel to start with

buf[0 or 1] retrieves same memory address

buf[0 or 1] +1 skips the padding

∴ in1 or in2 is the 1st pixel of that row!

The buf[y%2] and buf[(y+1)%2] also swaps

error diffusion

y%2 and (y+1)%2 → circular swaps → saves space