Computer Graphics

Prof. Feng Liu

Fall 2021

http://www.cs.pdx.edu/~fliu/courses/cs447/

10/06/2021

Last Time

- Color spaces
- Image file formats

Today

- In-class lab session
- Color quantization
- Dithering
- Homework 1 due today
- Homework 2 available
  - due before class on Oct. 20

Color Quantization

- The problem of reducing the number of colors in an image with minimal impact on appearance
  - Extreme case: 24 bit color to black and white
  - Less extreme: 24 bit color to 256 colors, or 256 grays
- Sub problems:
  - Decide which colors to use in the output (if there is a choice)
  - Decide which of those colors should be used for each input pixel

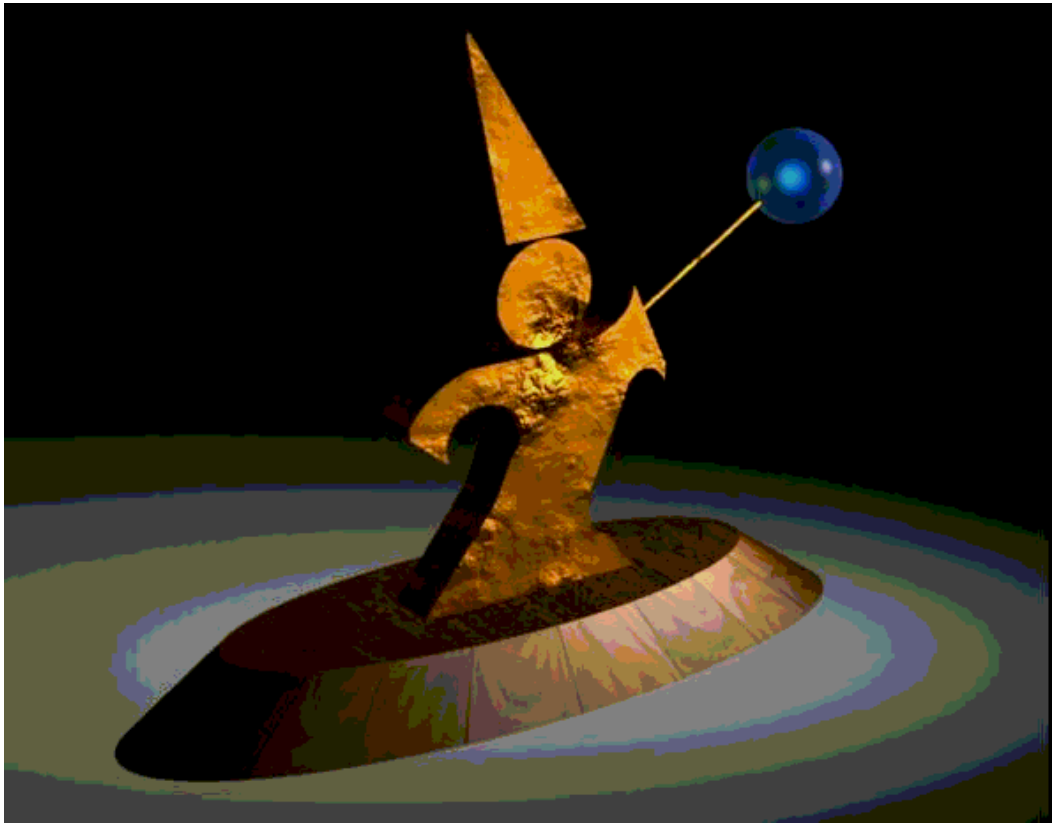
Example (24 bit color)

Uniform Quantization

- Break the color space into uniform cells
- Find the cell that each color is in, and map it to the center
- Equivalent to dividing each color by some number and taking the integer part
  - Say your original image is 24 bits color (8 red, 8 green, 8 blue)
  - Say you have 256 colors available, and you choose to use 8 reds, 8 greens and 4 blues ($8 \times 8 \times 4 = 256$)
  - Divide original red by 32, green by 32, and blue by 64
  - Some annoying details
- Generally does poorly because it fails to capture the distribution of colors
  - Some cells may be empty, and are wasted

Uniform Quantization

- 8 bits per pixel in this image
- Note that it does very poorly on smooth gradients
- Normally the hardest part to get right, because lots of similar colors appear very close together
- Does this scheme use information from the image?



Populosity Algorithm

- Build a color histogram: count the number of times each color appears
- Choose the *n* most commonly occurring colors
  - Typically group colors into *small* cells first using uniform quantization
- Map other colors to the closest chosen color
- Problem?

Histogram

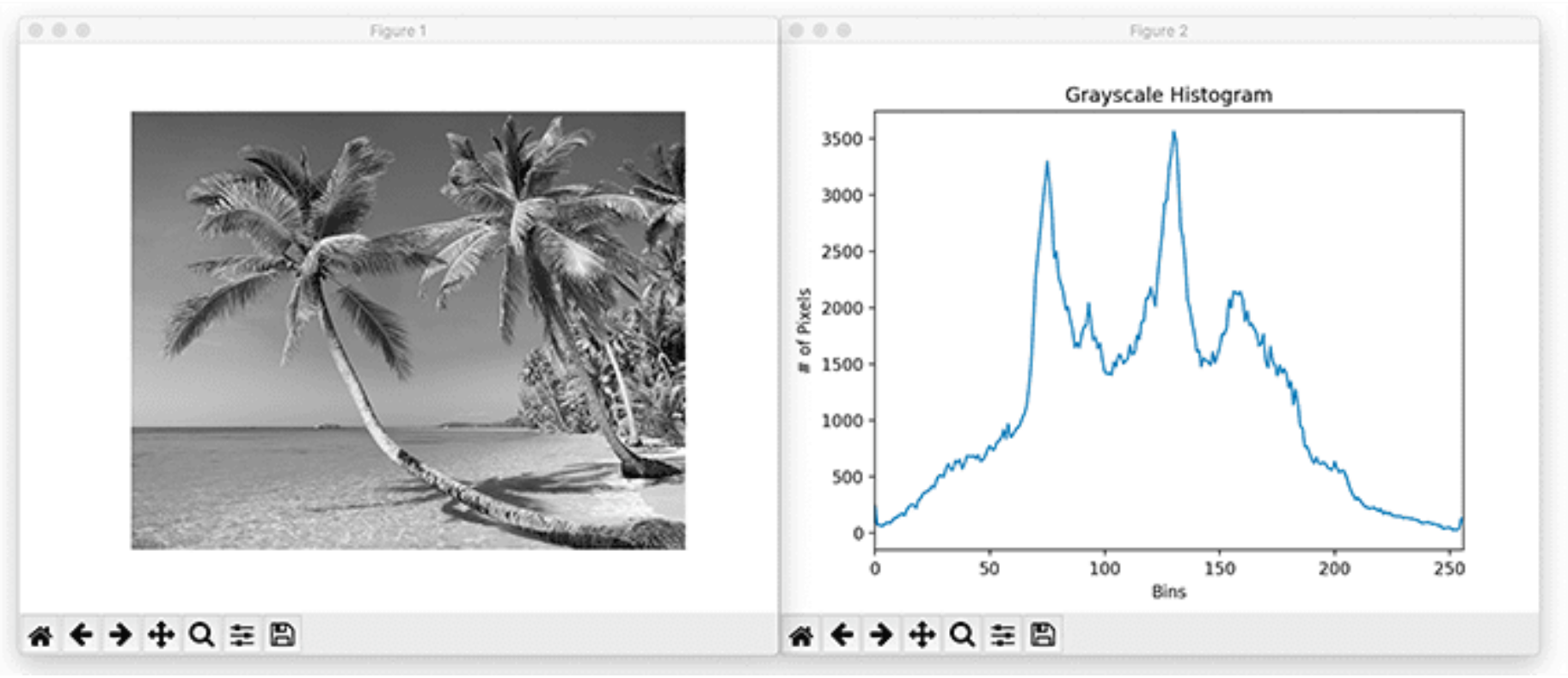


Image credit: https://www.pyimagesearch.com/2021/04/28/opencv-image-histograms-cv2-calchist/

Populosity Algorithm

- 8 bit image, so the most popular 256 colors

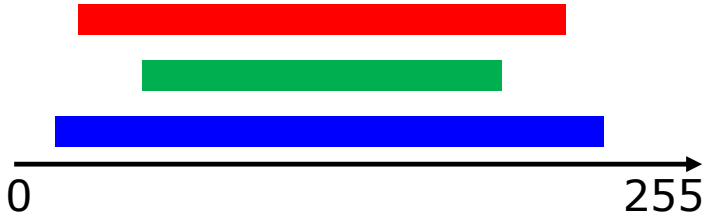
Populosity Algorithm

- 8 bit image, so the most popular 256 colors
- Note that blue wasn't very popular, so the crystal ball is now the same color as the floor
- Populosity ignores rare but important colors!

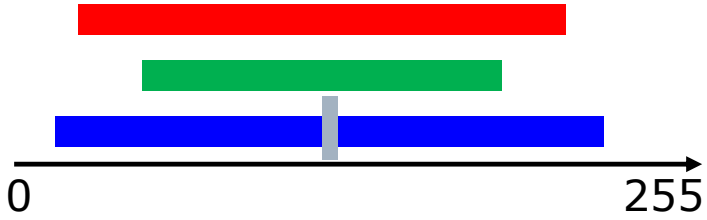
Median Cut (Clustering)

- View the problem as a *clustering* problem
  - Find groups of colors that are similar (a cluster)
  - Replace each input color with one representative of its cluster
- Many algorithms for clustering
- *Median Cut* is one: recursively
  - Find the “longest” dimension (r, g, b are dimensions)
  - Choose the median of the long dimension as a color to use
  - Split into two sub-clusters along the median plane, and recurse on both halves
- Works very well in practice

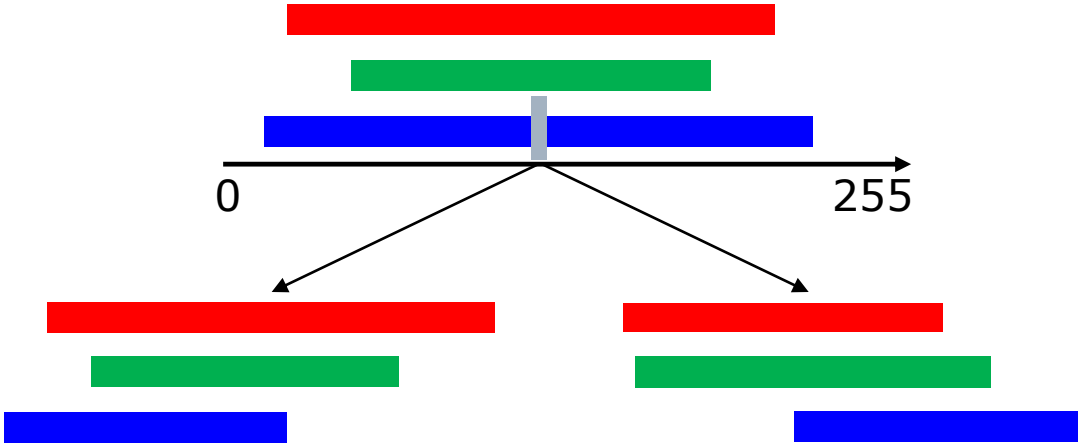
Median Cut (Clustering)



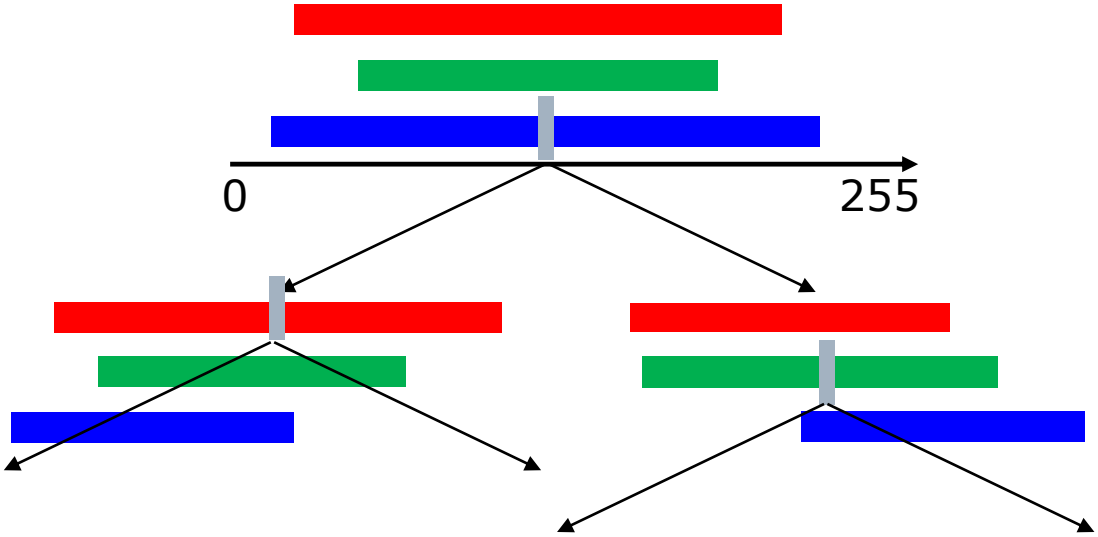
Median Cut (Clustering)



Median Cut (Clustering)



Median Cut (Clustering)



Median Cut

- 8 bit image, so 256 colors
- Now we get the blue
- Median cut works so well because it divides up the color space in the “most useful” way

Optimization Algorithms

- The quantization problem can be phrased as optimization
  - Find the set of colors and map that result in the lowest quantization error
- Several methods to solve the problem, but of limited use unless the number of colors to be chosen is small
  - It's expensive to compute the optimum
  - It's also a poorly behaved optimization

Perceptual Problems

- While a good quantization may get close colors, humans still perceive the quantization
- Biggest problem: *Mach bands*
  - The difference between two colors is more pronounced when they are side by side and the boundary is smooth
  - This emphasizes boundaries between colors, even if the color difference is small
  - Rough boundaries are “averaged” by our vision system to give smooth variation

Mach Bands in Reality

The floor appears banded

Mach Bands in Reality

Still some banding even in this 24 bit image (the floor in the background)

Dithering (Digital Halftoning)

- Mach bands can be removed by adding noise along the boundary lines
- General perceptive principle: replaced structured errors with noisy ones and people complain less
- Old industry dating to the late 1800's
  - Methods for producing grayscale images in newspapers and books

Today

- In-class lab session
- Color quantization
- Dithering
- Homework 1 due today
- Homework 2 available
  - due before class on Oct. 20

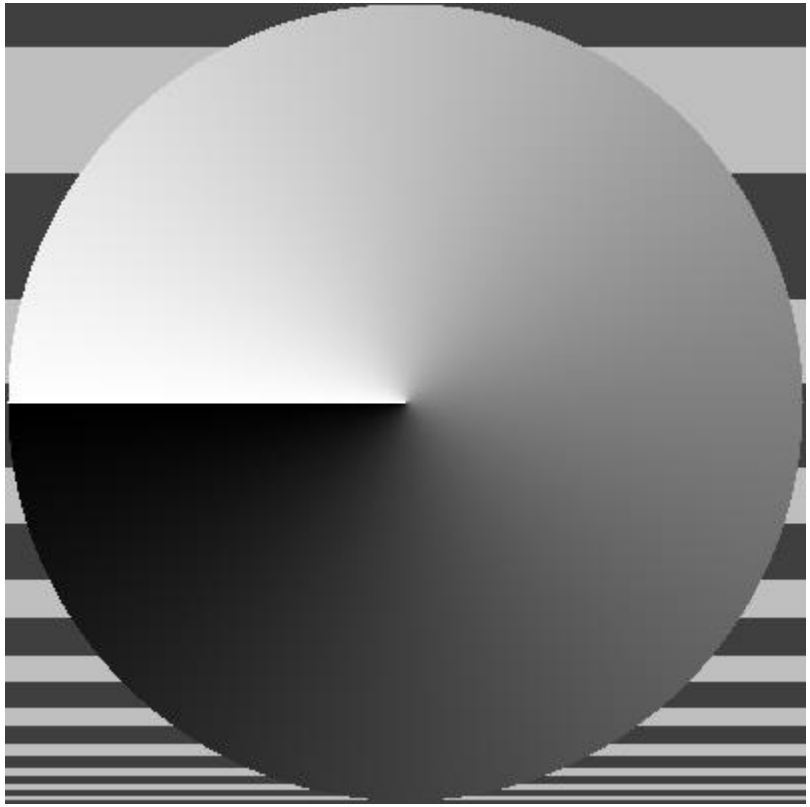
Dithering to Black-and-White

- Black-and-white is still the preferred way of displaying images in many areas
  - Black ink is cheaper than color
  - Printing with black ink is simpler and hence cheaper
  - Paper for black inks is not special

Dithering to Black-and-White

- To get color to black and white, first turn into grayscale: I=0.299R+0.587G+0.114B
  - This formula reflects the fact that green is more representative of perceived brightness than blue is
  - NOTE that it is **not** the equation implied by the RGB->XYZ color space conversion matrix
- **For all dithering we will assume that the image is gray and that intensities are represented as a value in [0, 1.0)**
  - Define new array of floating point numbers
  - `new_image[i] = old_image[i] / (float)256;`
  - To get back:
    `output[i]=(unsigned char)floor(new_image[i]*256)`

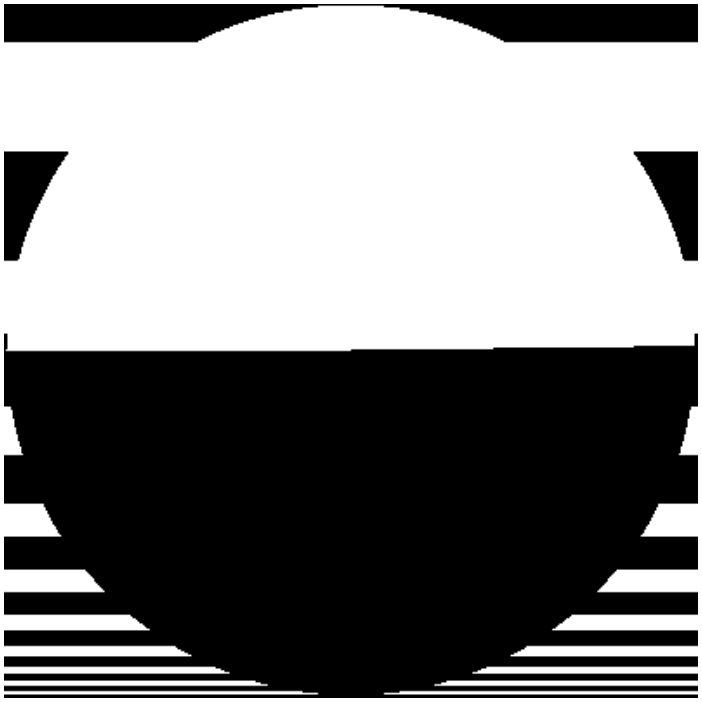
Sample Images



Threshold Dithering

- For every pixel: If the intensity < 0.5, replace with black, else replace with white
  - 0.5 is the threshold
  - This is the naïve version of the algorithm

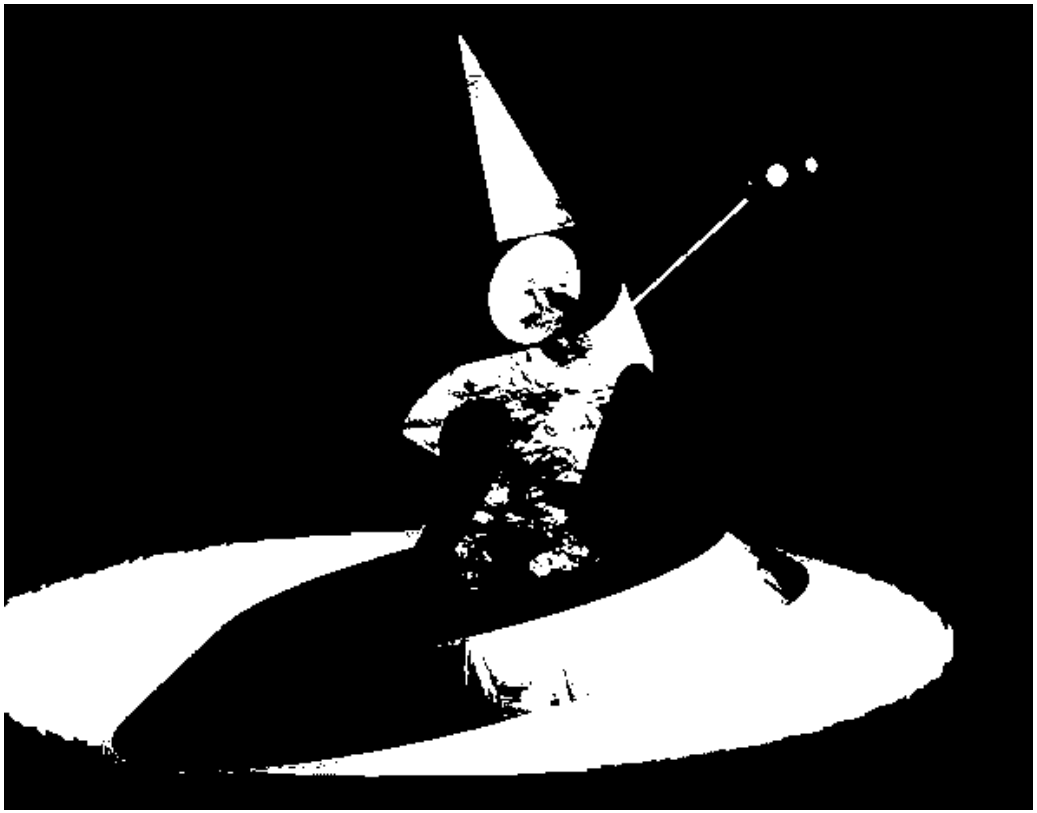
Naïve Threshold Algorithm

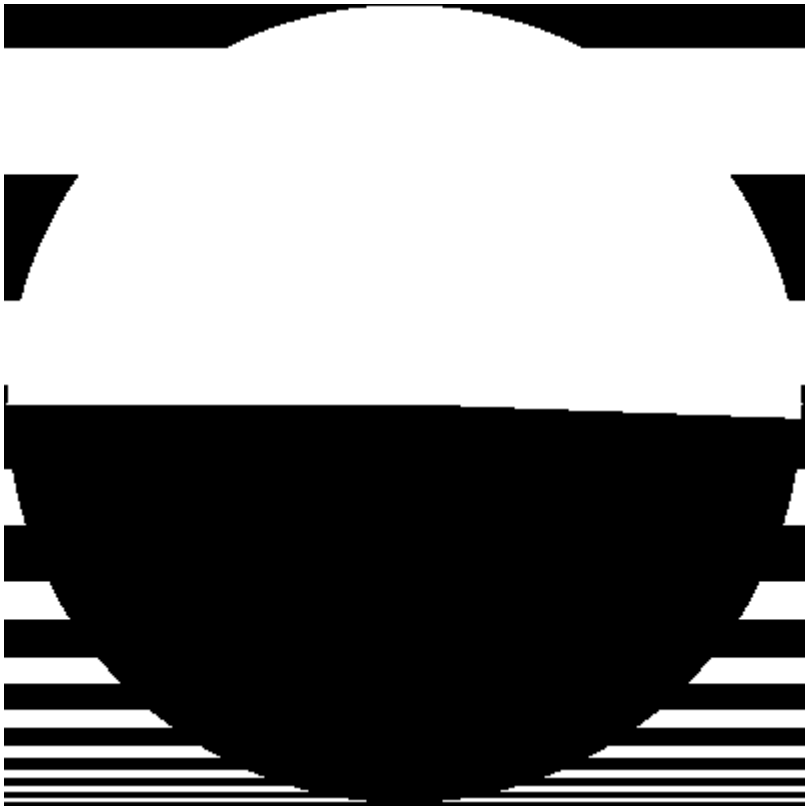


Threshold Dithering

- For every pixel: If the intensity < 0.5, replace with black, else replace with white
  - 0.5 is the threshold
  - This is the naïve version of the algorithm
- To keep the overall image brightness the same, you should:
  - Compute the average intensity over the image
  - Use a threshold that gives that average
  - For example, if the average intensity is 0.6, use a threshold that is higher than 40% of the pixels, and lower than the remaining 60%

Brightness Preserving Algorithm

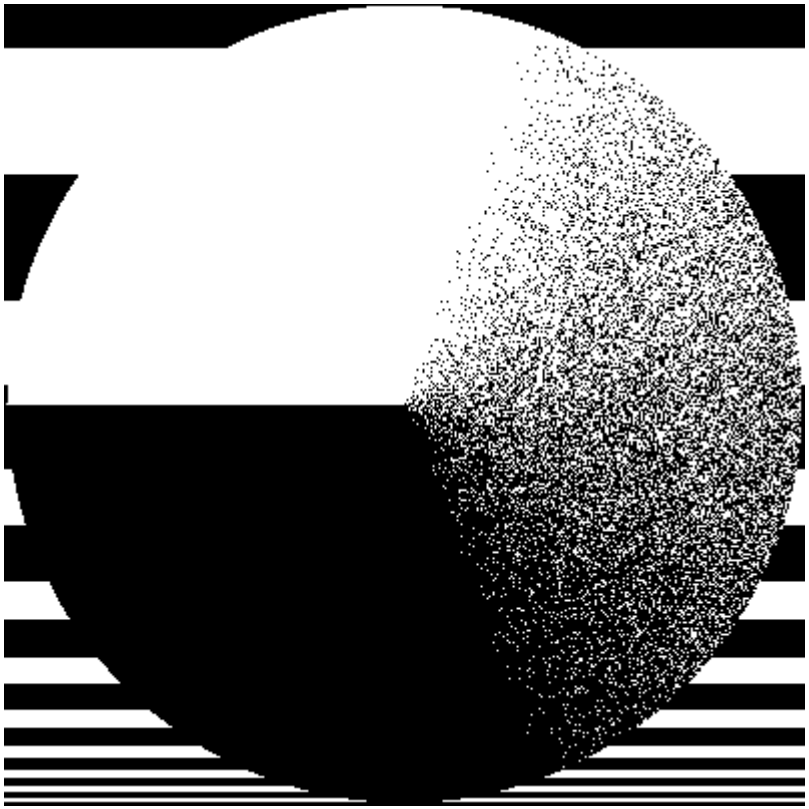




Random Modulation

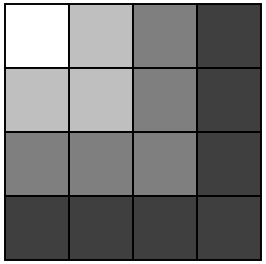
- Add a random amount to each pixel *before* thresholding
  - Typically add *uniformly* random amount from [-a,a]
- Pure addition of noise to the image
  - For better results, add better quality noise
  - For instance, use Gaussian noise (random values sampled from a normal distribution)
- Should use same procedure as before for choosing threshold
- Not good for black and white, but OK for more colors
  - Add a small random color to each pixel before finding the closest color in the table

Random Modulation



Ordered Dithering

- Break the image into small blocks
- Define a *threshold matrix*
  - Use a different threshold for each pixel of the block
  - Compare each pixel to its own threshold
- The thresholds can be clustered, which looks like newsprint
- The thresholds can be "random" which looks better

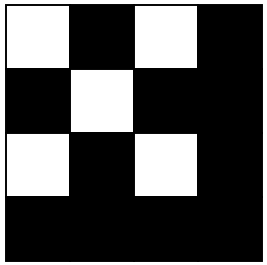


$$\begin{bmatrix} 1 & 0.75 & 0.5 & 0.25 \\ 0.75 & 0.75 & 0.5 & 0.25 \\ 0.5 & 0.5 & 0.5 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$$

Threshold matrix

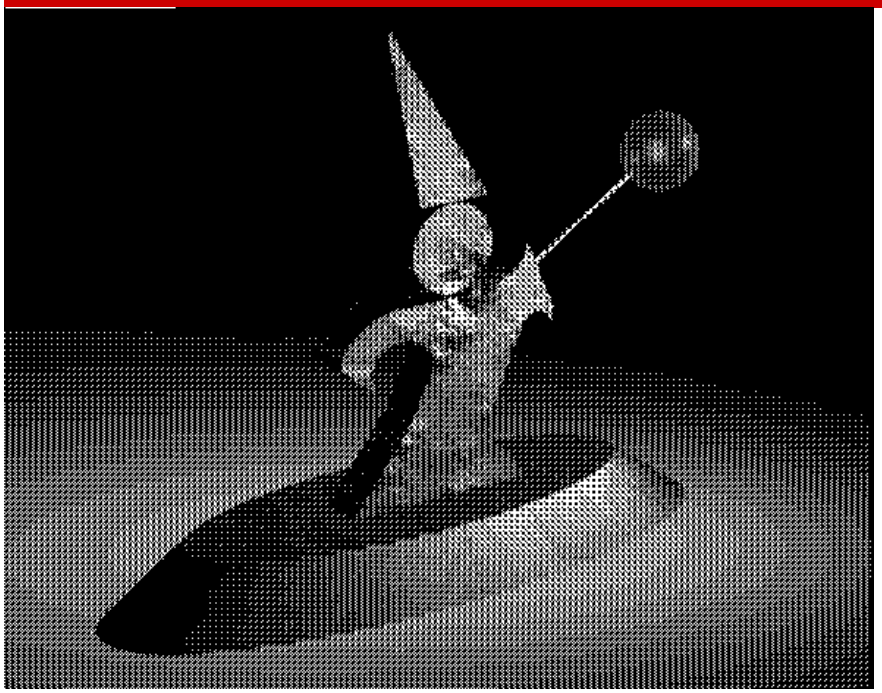
$$\frac{1}{16}\begin{bmatrix} 2 & 16 & 3 & 13 \\ 10 & 6 & 11 & 7 \\ 4 & 14 & 1 & 15 \\ 12 & 8 & 9 & 5 \end{bmatrix}$$

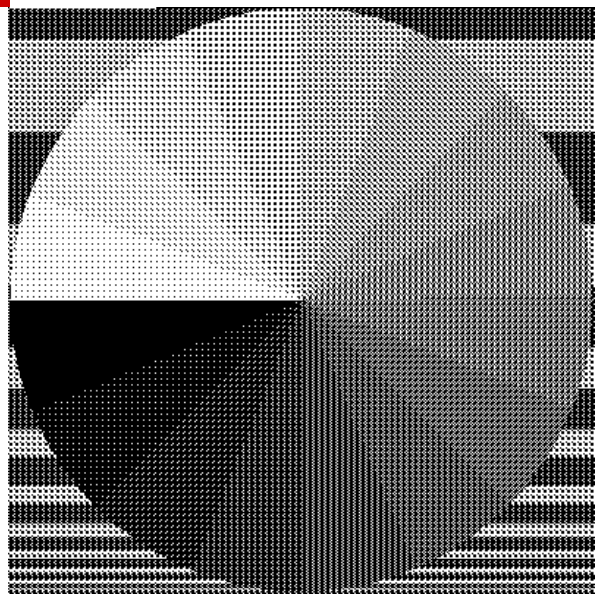
Result



$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

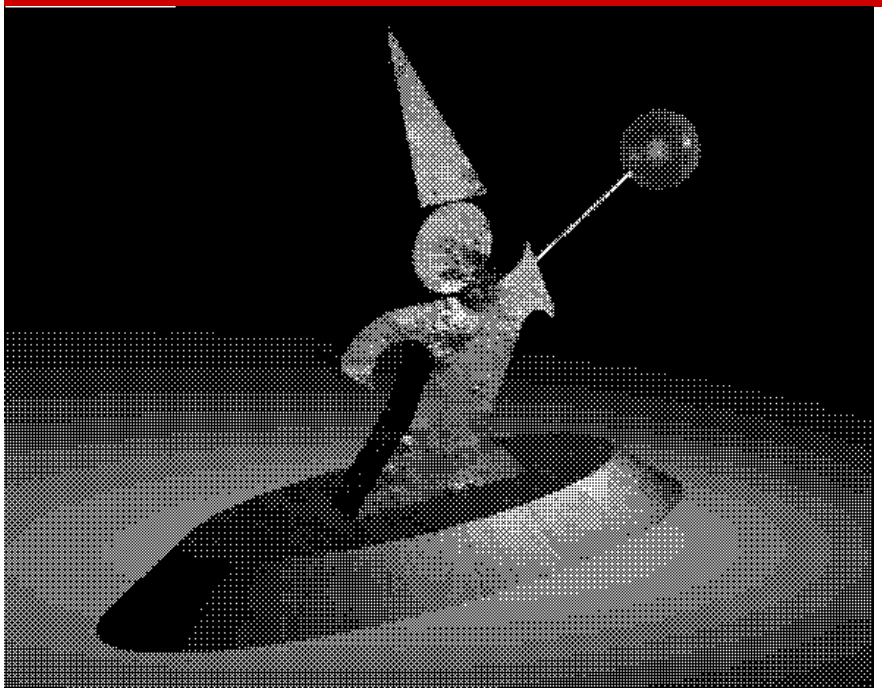
Clustered Dithering

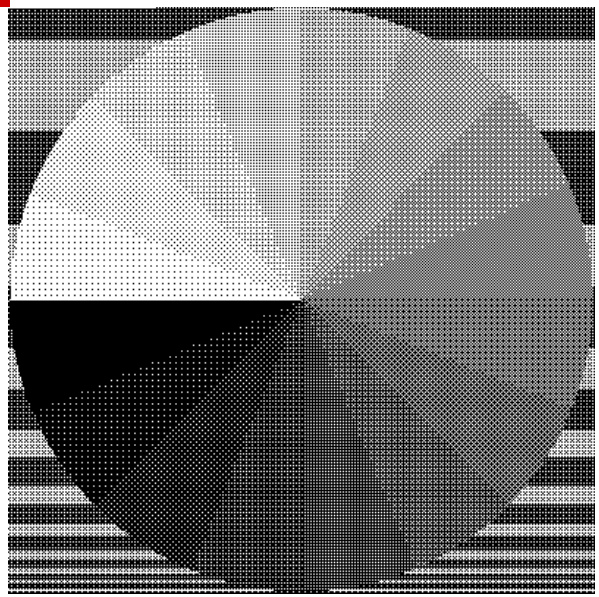




.75	.375	.625	.25
.0625	1	.875	.4375
.5	.8125	.9375	.125
.1875	.5625	.3125	.6875

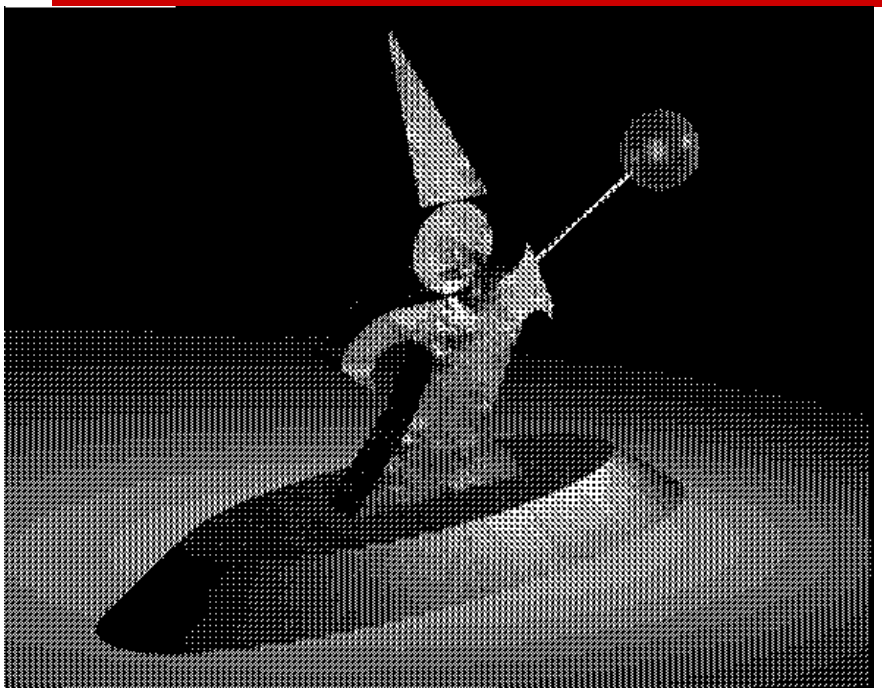
Dot Dispersion



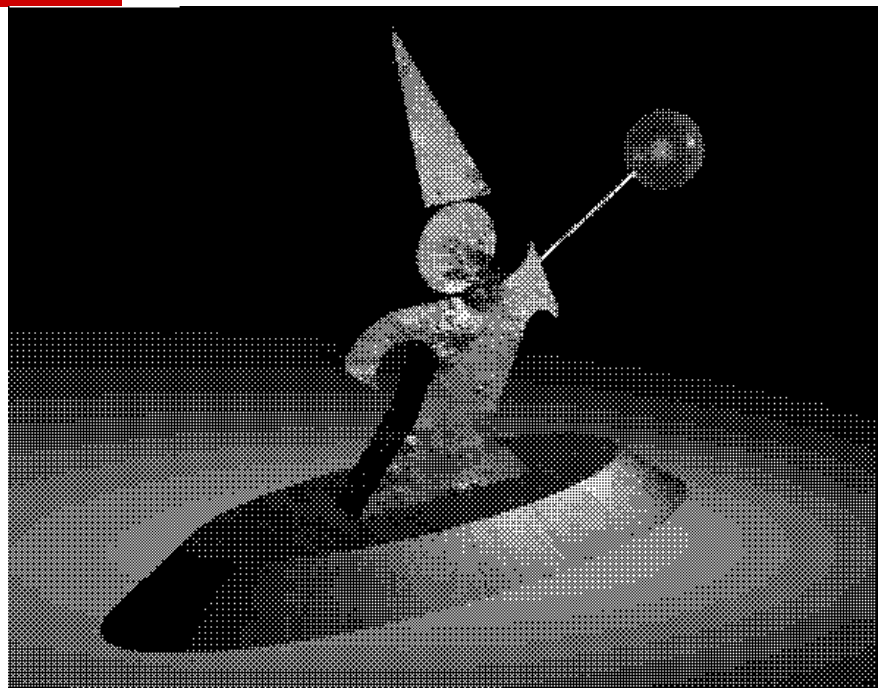


$$\begin{bmatrix} .125 & 1 & .1875 & .8125 \\ .625 & .375 & .6875 & .4375 \\ .25 & .875 & .0625 & .9375 \\ .75 & .5 & .5625 & .3125 \end{bmatrix}$$

Comparison



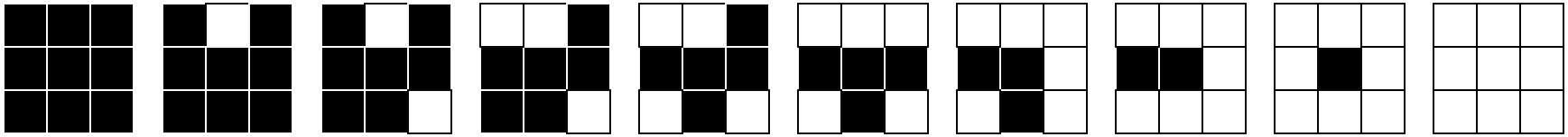
Clustered



Dot Dispersion

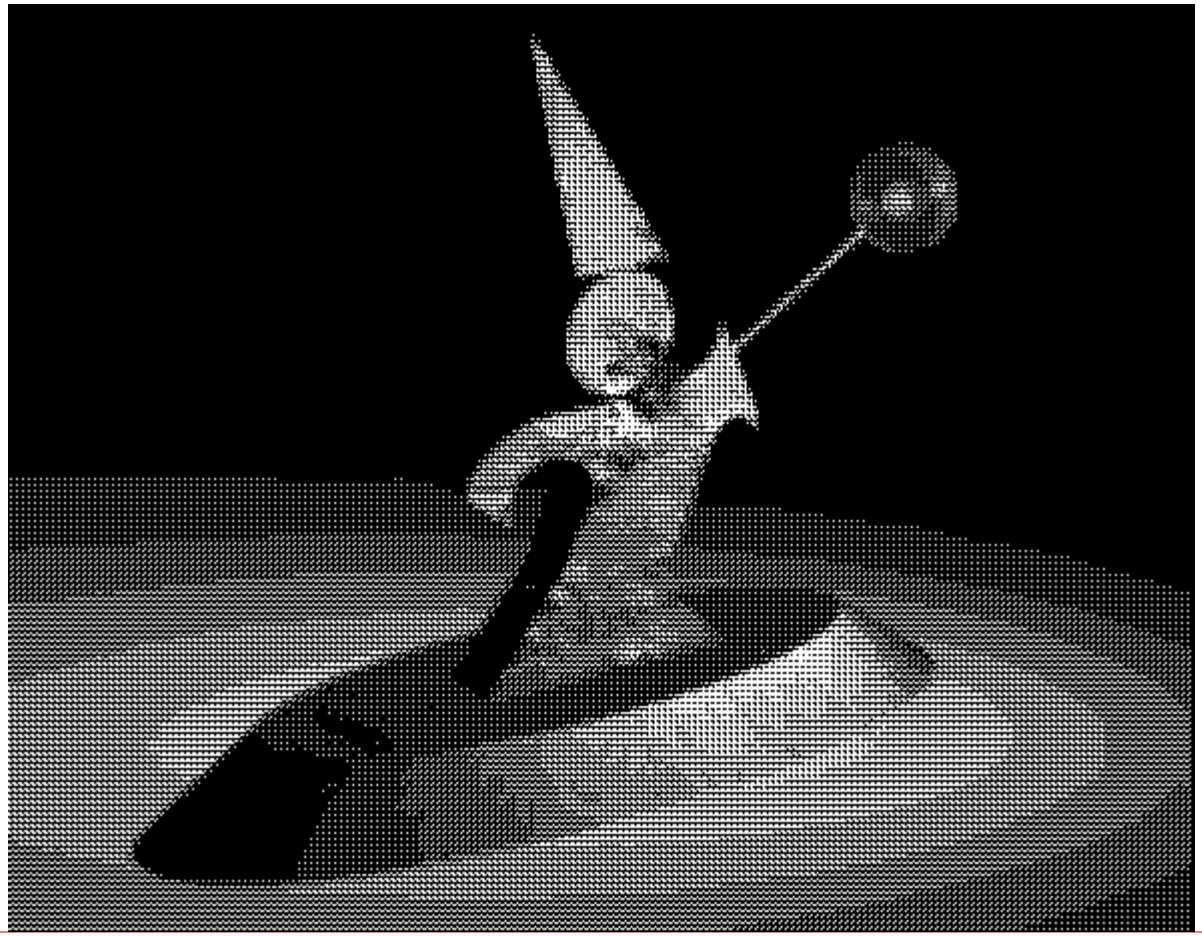
Pattern Dithering

- Compute the intensity of each sub-block and index a pattern

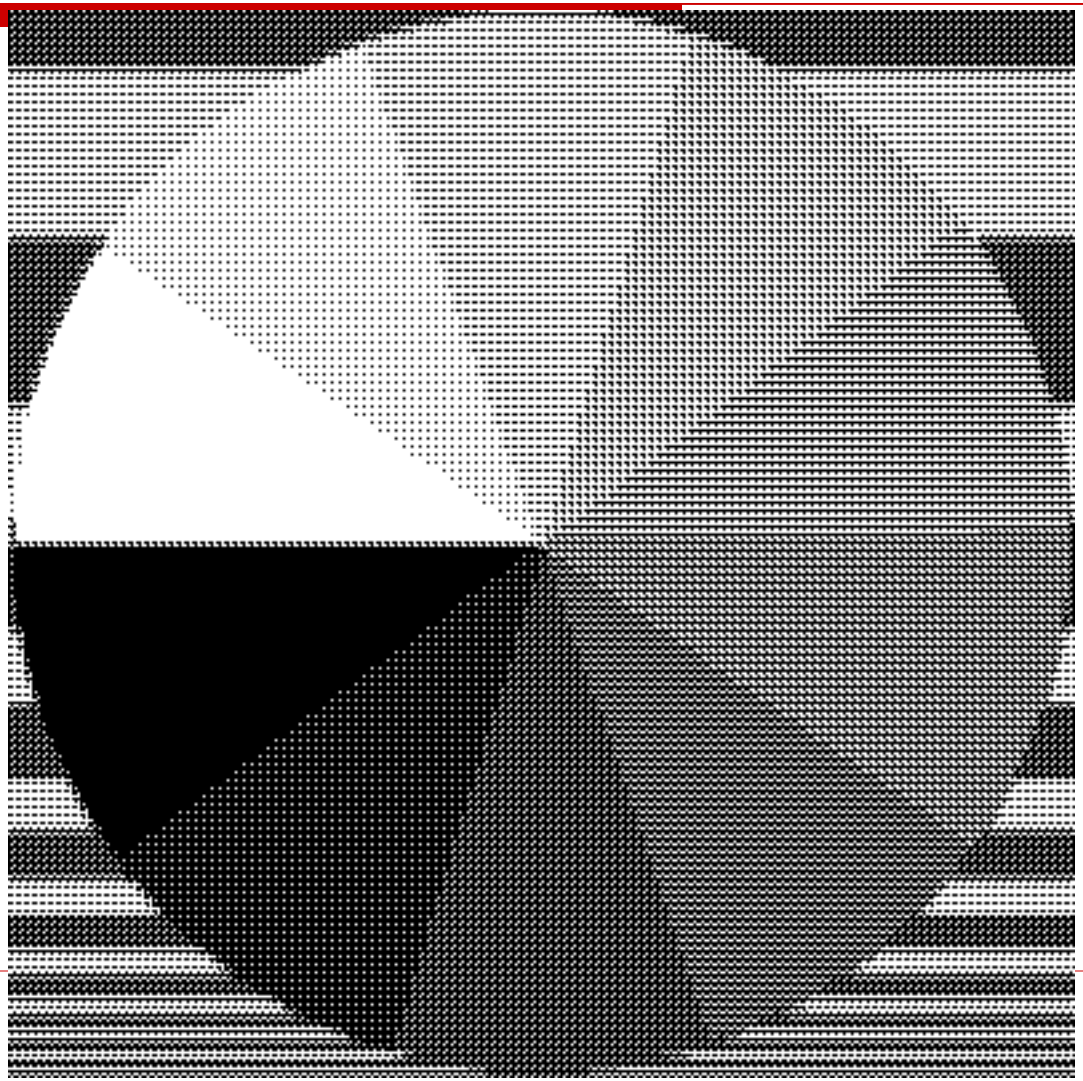


- NOT the same as before
  - Here, each sub-block has one of a fixed number of patterns – pixel is determined only by average intensity of sub-block
  - In ordered dithering, each pixel is checked against the dithering matrix before being turned on
- Used when display resolution is higher than image resolution – not uncommon with printers
  - Use 3x3 output for each input pixel

Pattern Dither (1)

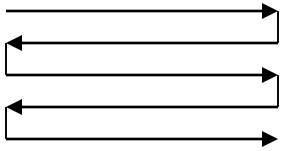


Pattern Dither (2)

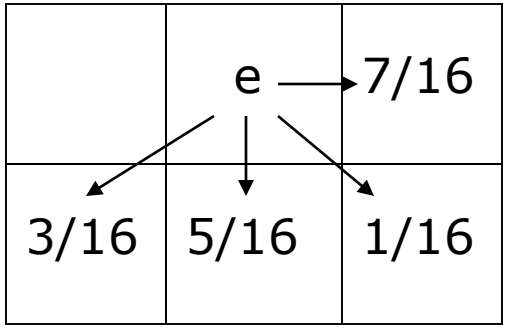


Floyd-Steinberg Dithering

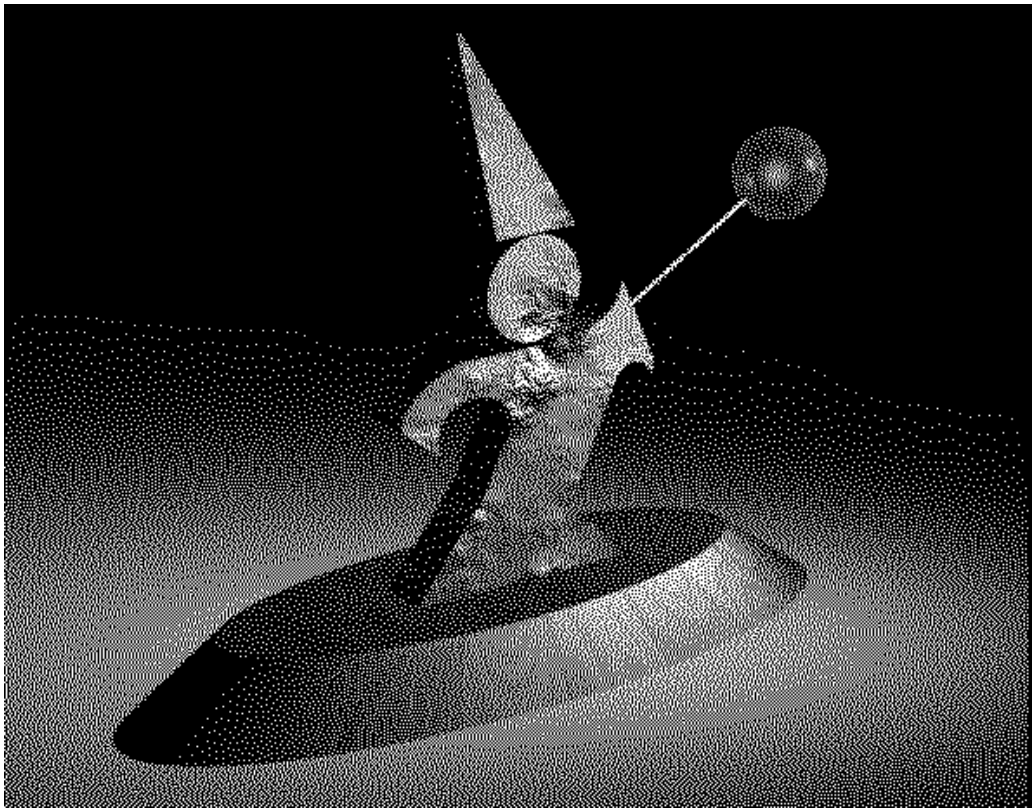
- Start at one corner and work through image pixel by pixel
  - Usually scan top to bottom in a zig-zag

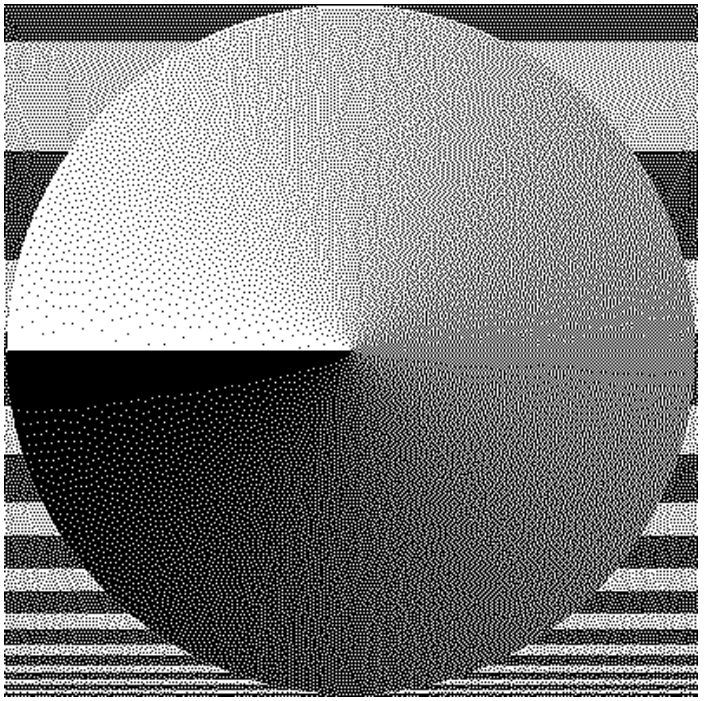


- Threshold each pixel
- Compute the error at that pixel: The difference between what should be there and what you did put there
  - If you made the pixel 0, *e* = original; if you made it 1, *e* = original-1
- Propagate error to neighbors by adding some proportion of the error to each unprocessed neighbor
  - A *mask* tells you how to distribute the error
- Easiest to work with floating point image
  - Convert all pixels to 0-1 floating point
- More detail in class reading materials



Floyd-Steinberg Dithering





Color Dithering

- All the same techniques can be applied, with some modification
- Example is Floyd-Steinberg:
  - Uniform color table
  - Error is difference from nearest color in the color table
  - Error propagation same as that for greyscale
    - Each color channel treated independently

Color Dithering

Comparison to Uniform Quant.

Same color table!

Next Time

- Signal processing
- Filtering
- Resampling