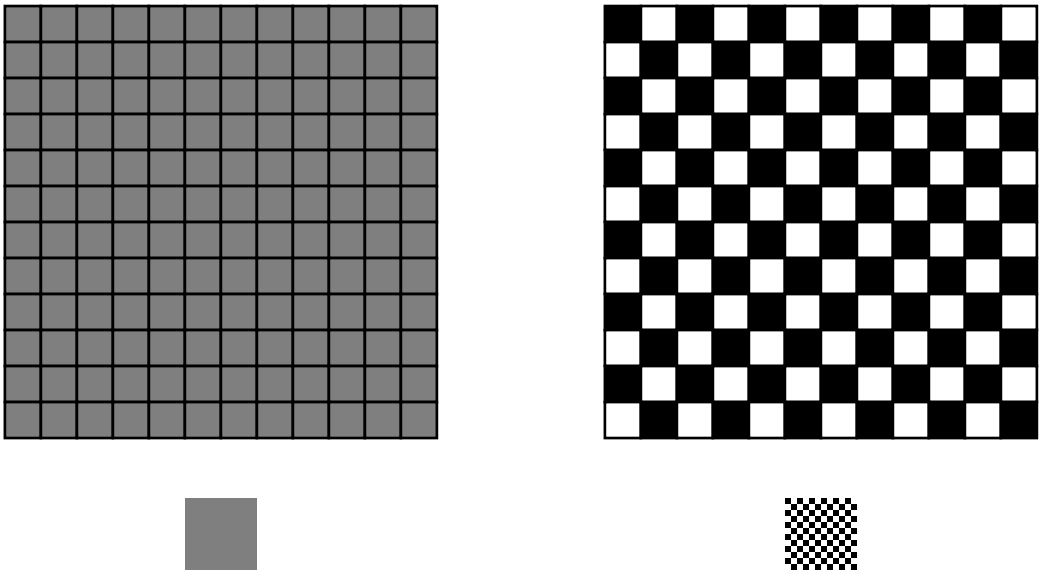
CS 559: Computer Graphics

Floyd-Steinberg Dithering

The Floyd-Steinberg dithering algorithm is an example of an error-diffusion technique. The aim is to use simple threshold dithering on each pixel, but to accurately account for the errors in brightness it induces. As a simple motivating example, consider a 50% gray image (an image with every pixel exactly halfway between black and white in brightness). We would like the dithered image to end up half black and half white, ideally with a black-white checker pattern at the pixel level (as in the image below). This most accurately captures the 50% gray aspect using only black and white pixel intensities. This cannot be done with simple thresholding, because in a 50% gray image every pixel starts with the same intensity, so the result of a thresholding comparison must be the same for every pixel.



The solution is to take into account the result of thresholding one pixel when thresholding its neighbors. This is the essence of an error-diffusion algorithm: the error as a result of thresholding one pixel is shifted to its neighbors where it influences their threshold operation.

First we must define the error, e. This is the difference between the target value of a pixel, $I_{acc}(i,j)$ and its value after thresholding, $I_{out}(i,j)$. Specifically, $e = I_{acc}(i,j) - t$ where $t = 0$ if $I_{acc}(i,j) < 0.5$ and $t = 1$ if $I_{acc}(i,j) \geq 0.5$. I_{acc} is the accumulated image. It starts off as the input gray image in floating point format with pixel values ranging from 0 (black) to 1 (white). At each step, the error, e, from one pixel is added to its neighbors using the following algorithm:

$$
\begin{aligned}
I_{acc}(i+1,j) &= I_{acc}(i+1,j) + \frac{7}{16}e \\
I_{acc}(i+1,j+1) &= I_{acc}(i+1,j+1) + \frac{3}{16}e \\
I_{acc}(i,j+1) &= I_{acc}(i,j+1) + \frac{5}{16}e \\
I_{acc}(i-1,j+1) &= I_{acc}(i-1,j+1) + \frac{1}{16}e
\end{aligned}
$$

(assuming that the index j increases from top to bottom of the image, although it really doesn't matter in practice). Note that the error accumulates in the image I_{acc}, and that the value in I_{acc} may

go above 1 or below 0. That doesn't matter. If one of the pixels in I_{acc} required above is off the edge of the image, we ignore it.

The pixel (i, j) is thresholded and output: $I_{out}(i, j) = 0$ if $I_{acc}(i, j) < 0.5$ or $I_{out}(i, j) = 1$ if $I_{acc}(i, j) \geq 0.5$.

The algorithm proceeds by performing the above step on every pixel (i, j) in a zig-zag order. All the even numbered rows (starting at 0) go left to right using the algorithm above. All the odd numbered rows go right to left, using $i - 1$ every place where you see $i + 1$ and vice versa. When it's all done, every pixel in I_{out} has been filled and the image I_{acc} can be thrown away.

In the set of images on the next page, we show the images I_{acc} and I_{out} for a 4×4 50%-gray image as the algorithm progresses.

I_{acc}

0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5

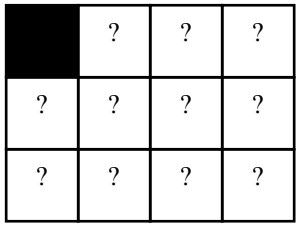
I_{out}

?	?	?	?
?	?	?	?
?	?	?	?

I_{acc}

X	0.719	0.5	0.5
0.656	0.531	0.5	0.5
0.5	0.5	0.5	0.5

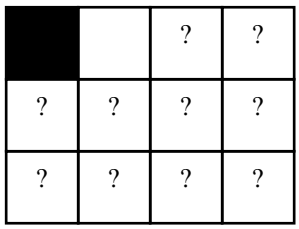
I_{out}



I_{acc}

X	X	0.377	0.5
0.604	0.443	0.482	0.5
0.5	0.5	0.5	0.5

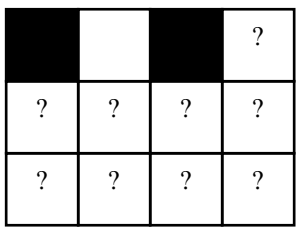
I_{out}



I_{acc}

X	X	X	0.665
0.604	0.514	0.6	0.524
0.5	0.5	0.5	0.5

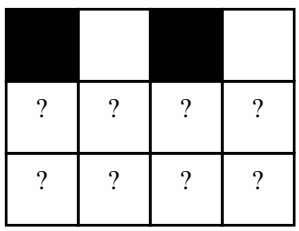
I_{out}



I_{acc}

X	X	X	X
0.604	0.514	0.537	0.419
0.5	0.5	0.5	0.5

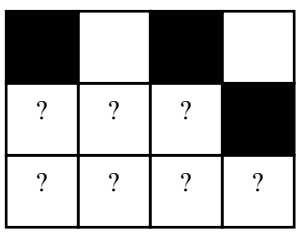
I_{out}



I_{acc}

X	X	X	X
0.604	0.514	0.721	X
0.5	0.5	0.526	0.631

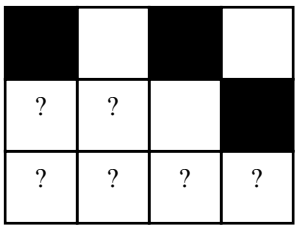
I_{out}



I_{acc}

X	X	X	X
0.604	0.392	X	X
0.5	0.483	0.439	0.579

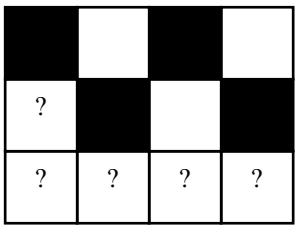
I_{out}



I_{acc}

X	X	X	X
0.775	X	X	X
0.524	0.605	0.512	0.579

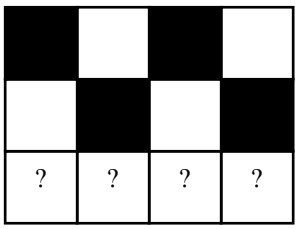
I_{out}



I_{acc}

X	X	X	X
X	X	X	X
0.454	0.563	0.512	0.579

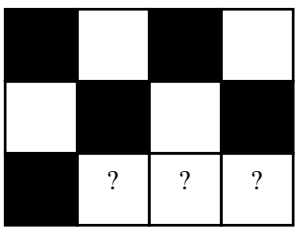
I_{out}



I_{acc}

X	X	X	X
X	X	X	X
X	0.761	0.512	0.579

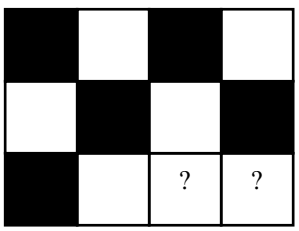
I_{out}



I_{acc}

X	X	X	X
X	X	X	X
X	X	0.408	0.579

I_{out}



I_{acc}

X	X	X	X
X	X	X	X
X	X	X	0.757

I_{out}