Image processing basics

Andy Warhol, *Marilyn Diptych*, 1962 (source)

Many slides adapted from
Alyosha Efros, Derek Hoiem

Image processing basics: Outline

- Images as sampled functions
- Sampling and reconstruction, aliasing
- Image resampling, interpolation
- Image transformations

Model of imaging

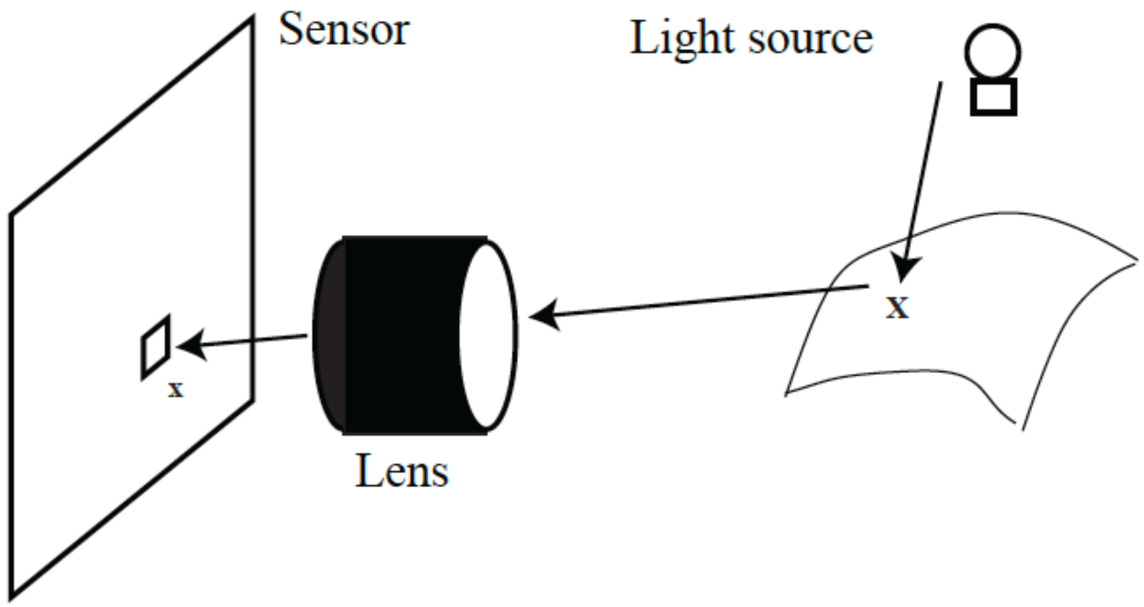
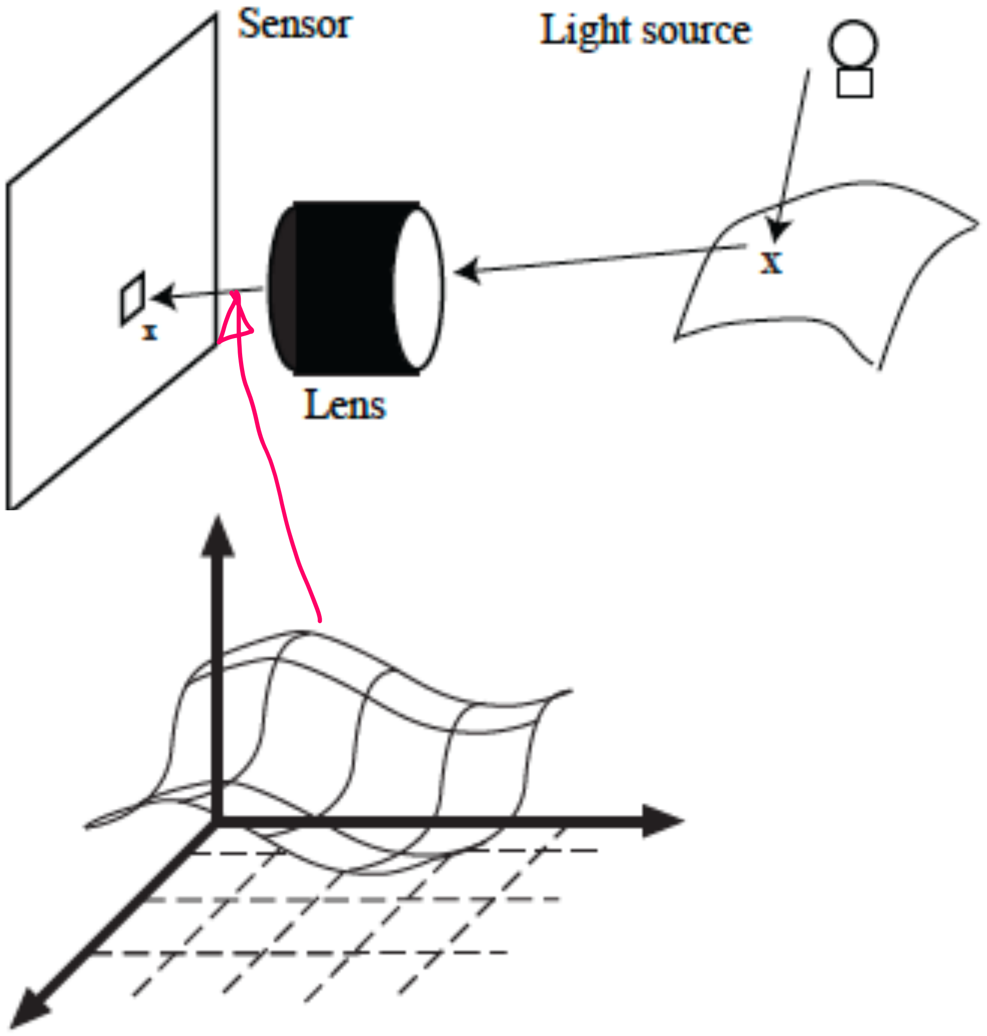


FIGURE 2.1: *A high-level model of imaging. Light leaves light sources and reflects from surfaces. Eventually, some light arrives at a camera and enters a lens system. Some of that light arrives at a photosensor inside the camera.*


Sensor
Light source
x
x
Lens

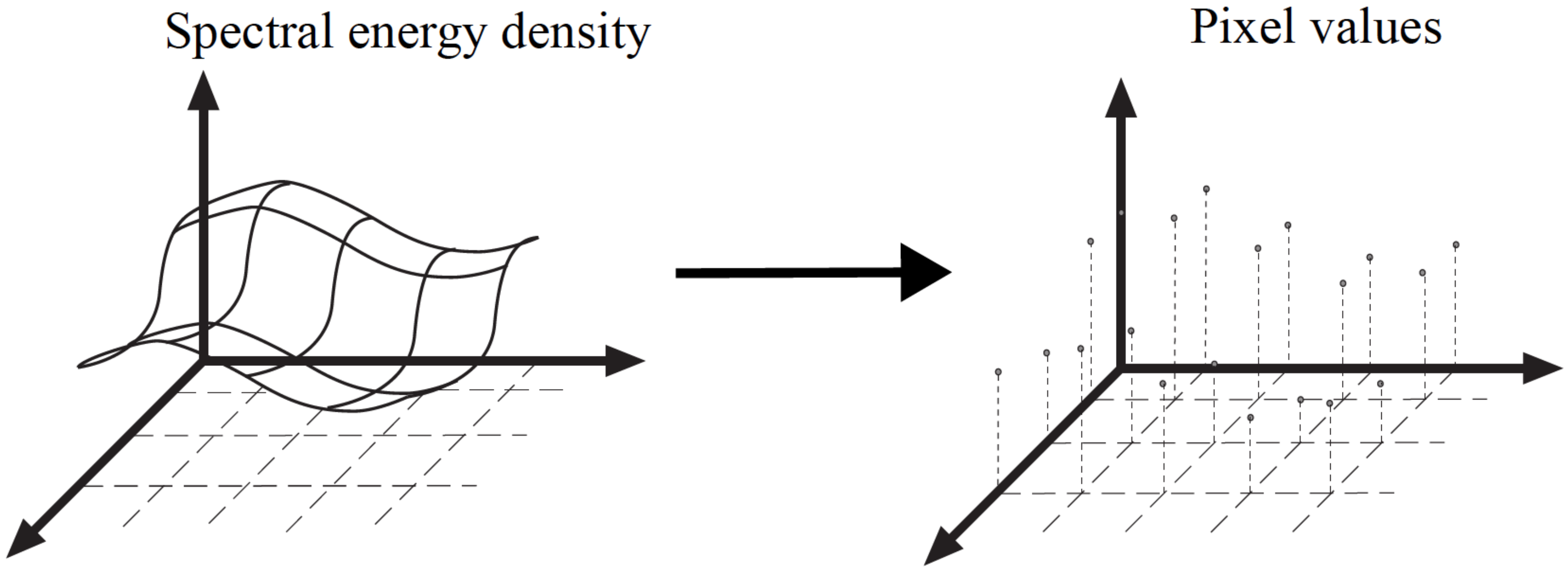
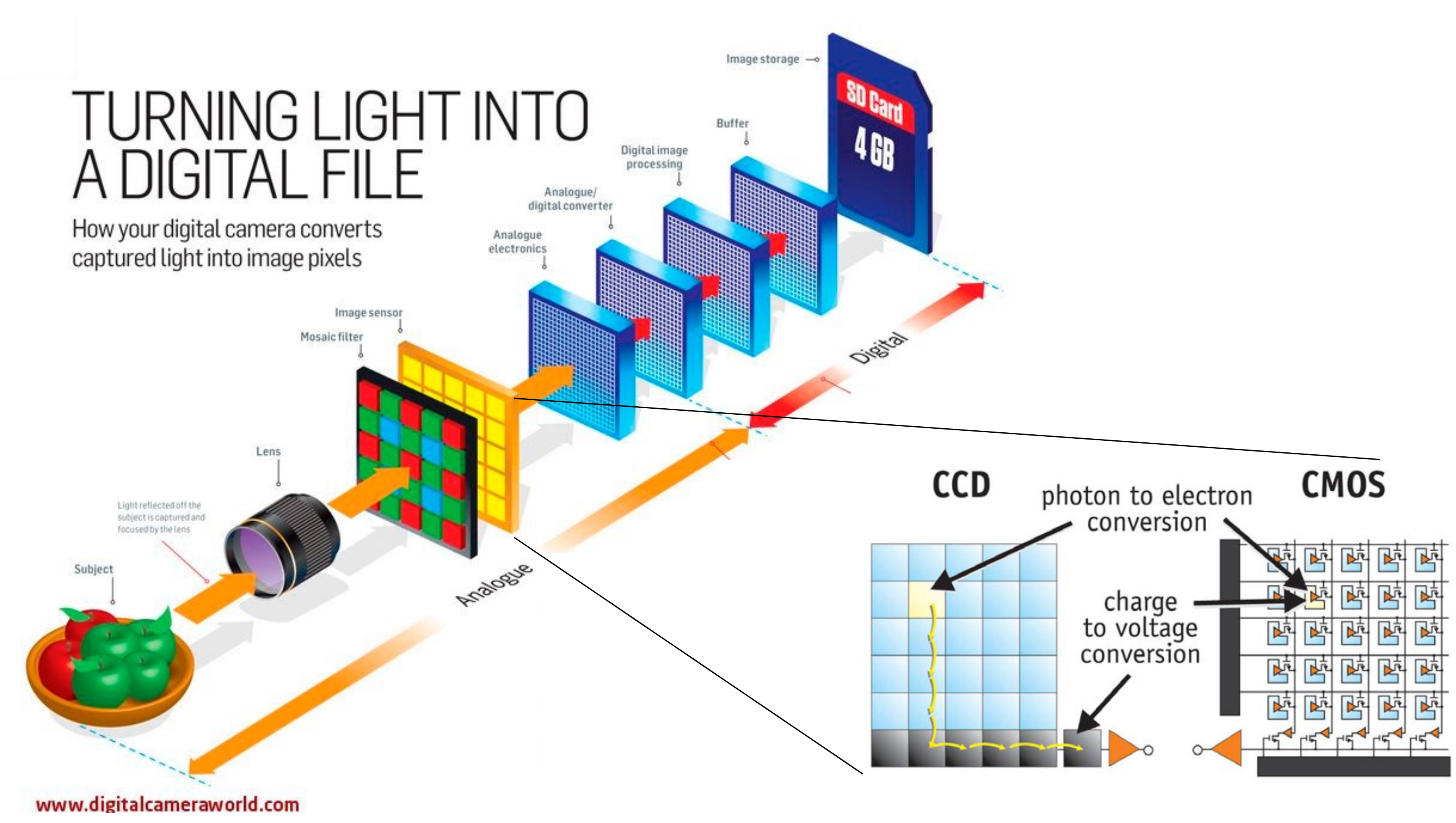


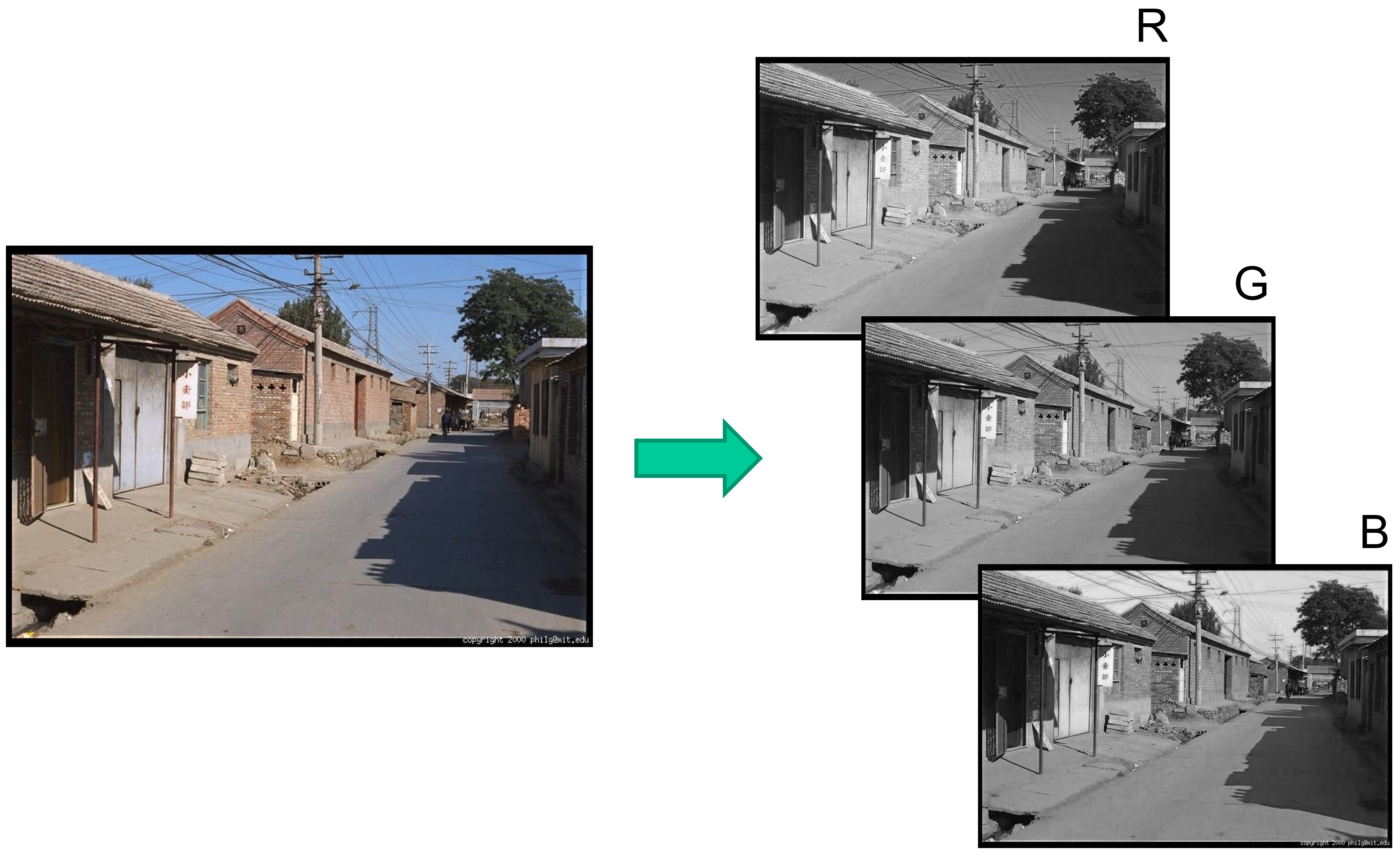
FIGURE 2.2: *Because each pixel in the sensor averages over a small range of directions and positions, the process mapping the input spectral energy distribution to pixel values can be thought of as sampling. On the* **left***, is a representation of the energy distribution as a continuous function of position. The value reported at each pixel is the value of this function at the location of the pixel (***right***).*

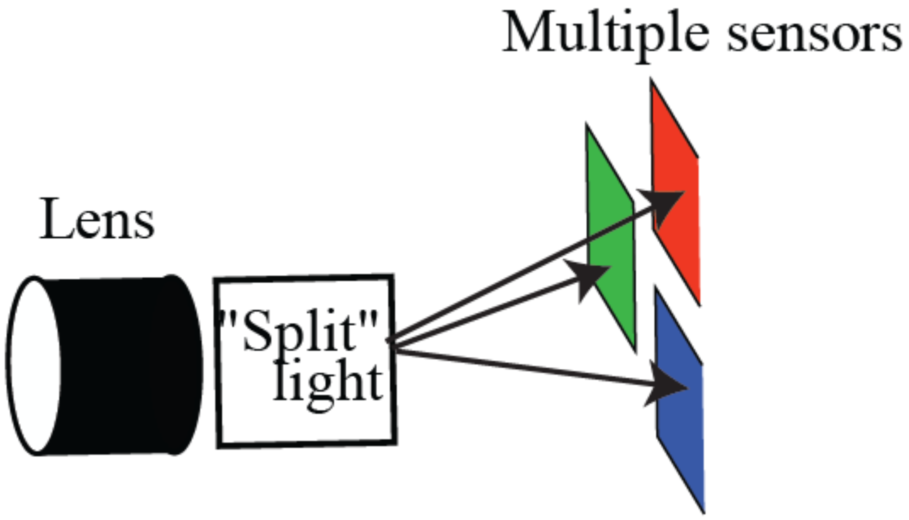
Images as sampled functions

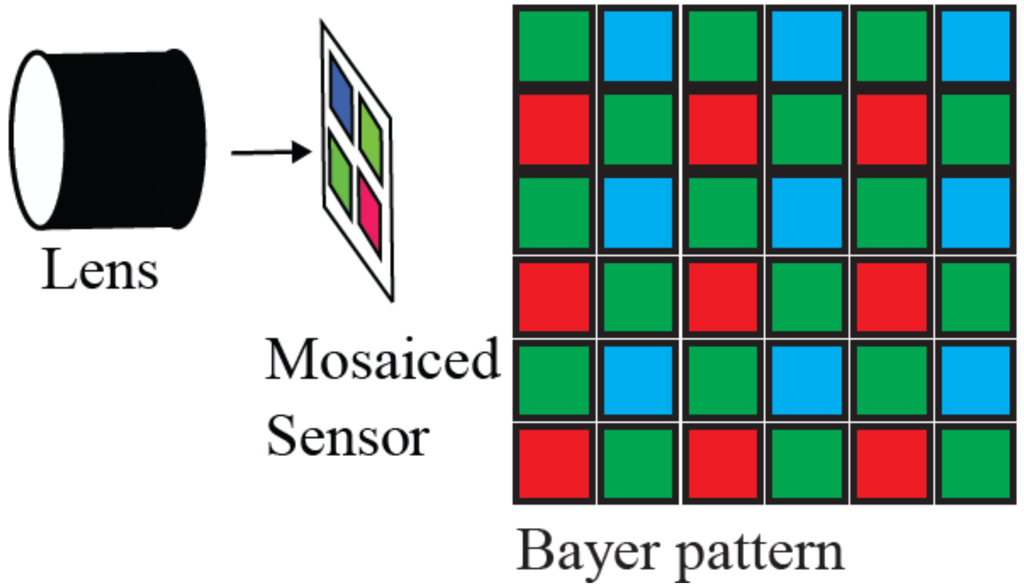


TURNING LIGHT INTO A DIGITAL FILE
How your digital camera converts captured light into image pixels
Image storage
SD Card
4 GB
Buffer
Digital image processing
Analogue/ digital converter
Analogue electronics
Image sensor
Mosaic filter
Lens
Light reflected off the subject is captured and focused by the lens
Subject
Digital
Analogue
www.digitalcameraworld.com
CCD
photon to electron conversion
CMOS
charge to voltage conversion

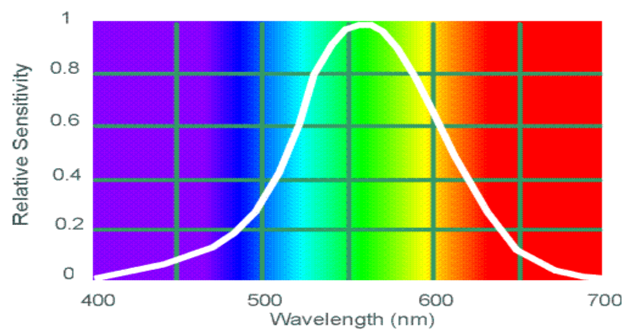
Digital color image







Why more green?



Human Luminance Sensitivity Function

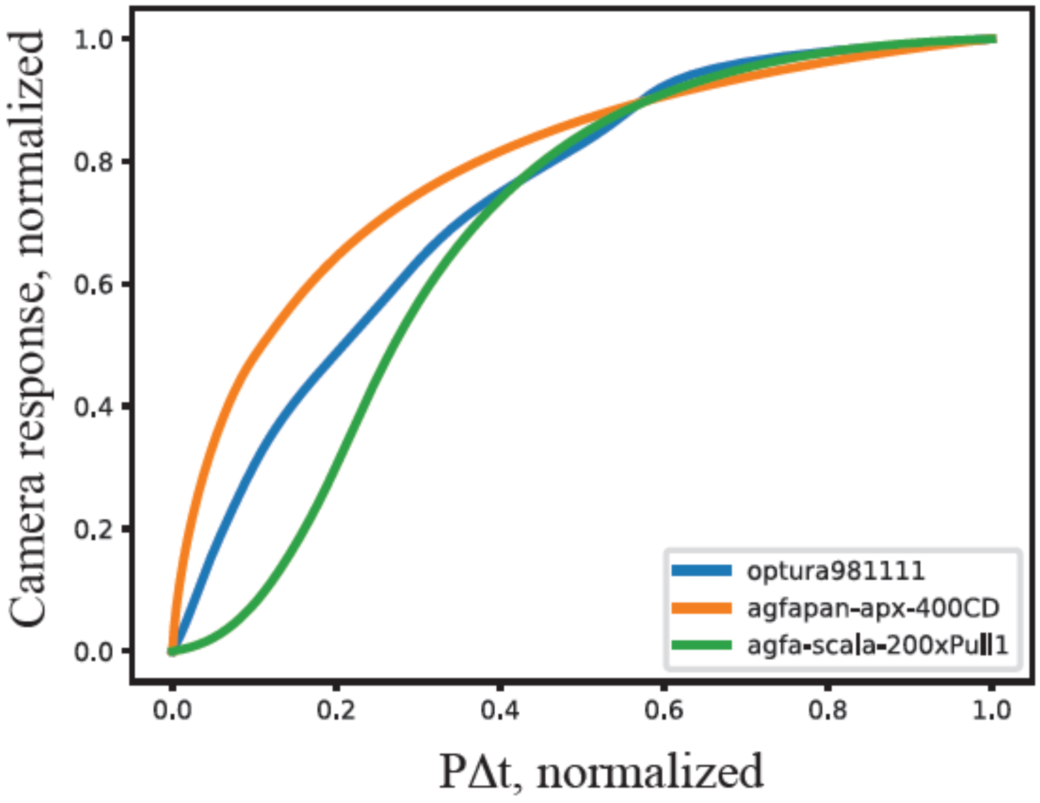
Cameras aren’t linear in input energy

This is deliberate,
and usually a property
of camera electronics.

Different cameras are
different

Helps imitate film

Increases dynamic range



Images in Python

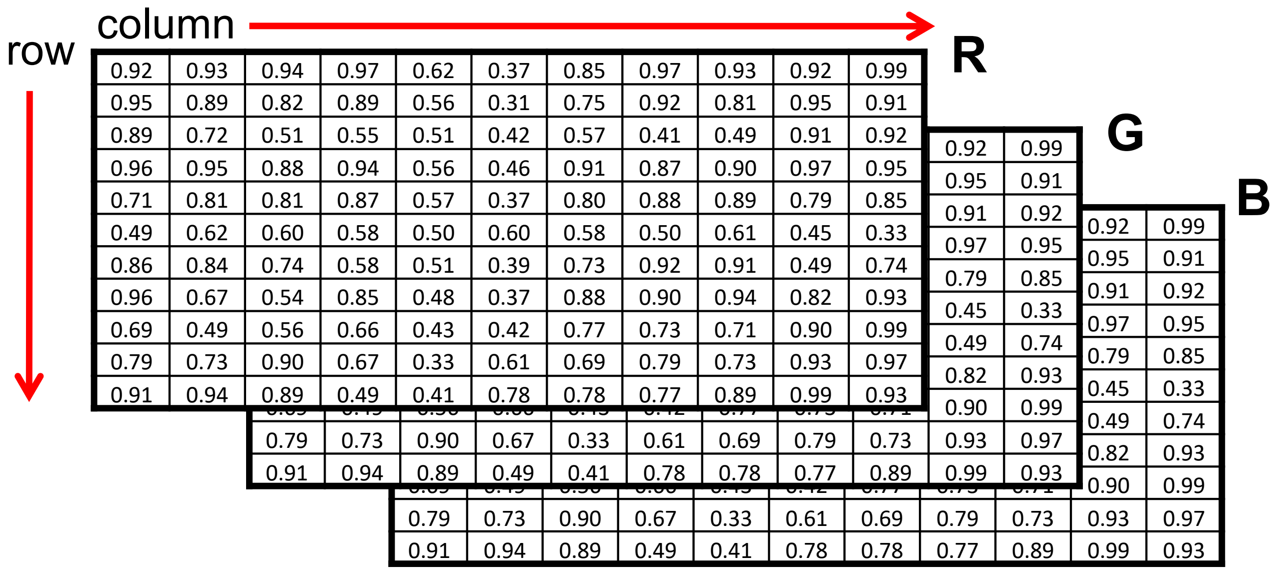
```
im = cv2.imread(filename)                 # read image
im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB) # order channels as RGB
im = im / 255                             # values range from 0 to 1
```

RGB image `im` is a H x W x 3 matrix (numpy.ndarray)

`im[0,0,0]` is the top-left pixel value in R-channel

`im[y, x, c]` is the value y+1 pixels down, x+1 pixels to right in the cth channel

`im[H-1, W-1, 2]` is the bottom-right pixel in B-channel

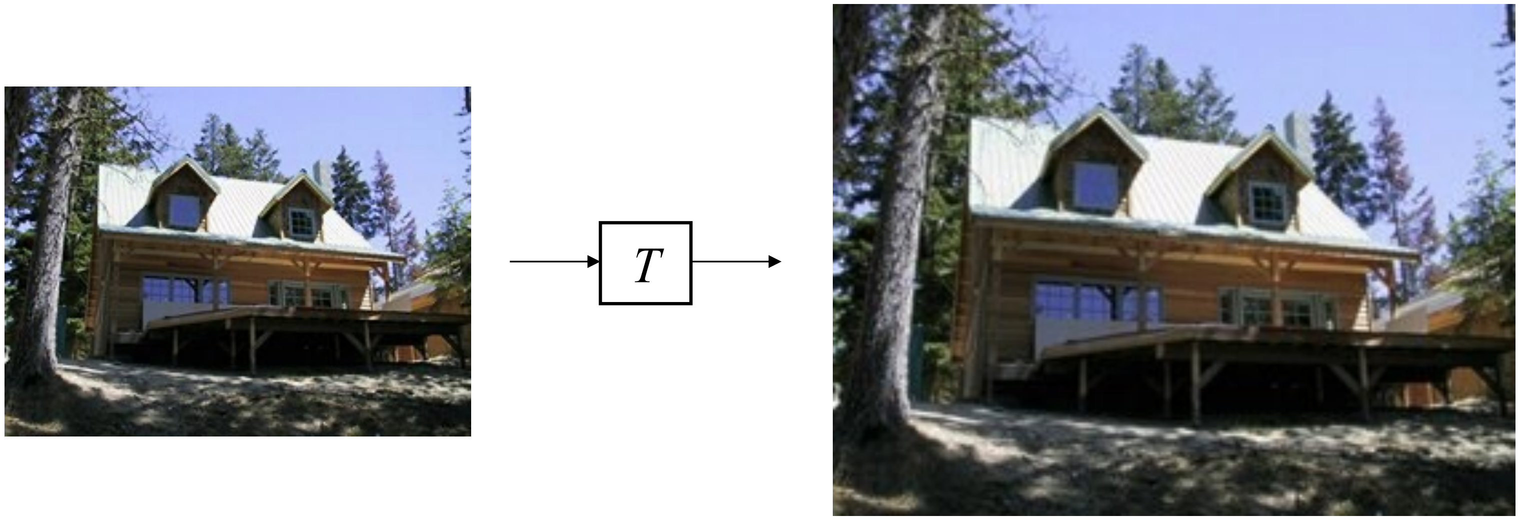


0.92	0.93	0.94	0.97	0.62	0.37	0.85	0.97	0.93	0.92	0.99
0.95	0.89	0.82	0.89	0.56	0.31	0.75	0.92	0.81	0.95	0.91
0.89	0.72	0.51	0.55	0.51	0.42	0.57	0.41	0.49	0.91	0.92
0.96	0.95	0.88	0.94	0.56	0.46	0.91	0.87	0.90	0.97	0.95
0.71	0.81	0.81	0.87	0.57	0.37	0.80	0.88	0.89	0.79	0.85
0.49	0.62	0.60	0.58	0.50	0.60	0.58	0.50	0.61	0.45	0.33
0.86	0.84	0.74	0.58	0.51	0.39	0.73	0.92	0.91	0.49	0.74
0.96	0.67	0.54	0.85	0.48	0.37	0.88	0.90	0.94	0.82	0.93
0.69	0.49	0.56	0.66	0.43	0.42	0.77	0.73	0.71	0.90	0.99
0.79	0.73	0.90	0.67	0.33	0.61	0.69	0.79	0.73	0.93	0.97
0.91	0.94	0.89	0.49	0.41	0.78	0.78	0.77	0.89	0.99	0.93

How are the three color channels acquired?

Image transformations

Upsampling



Upsampling

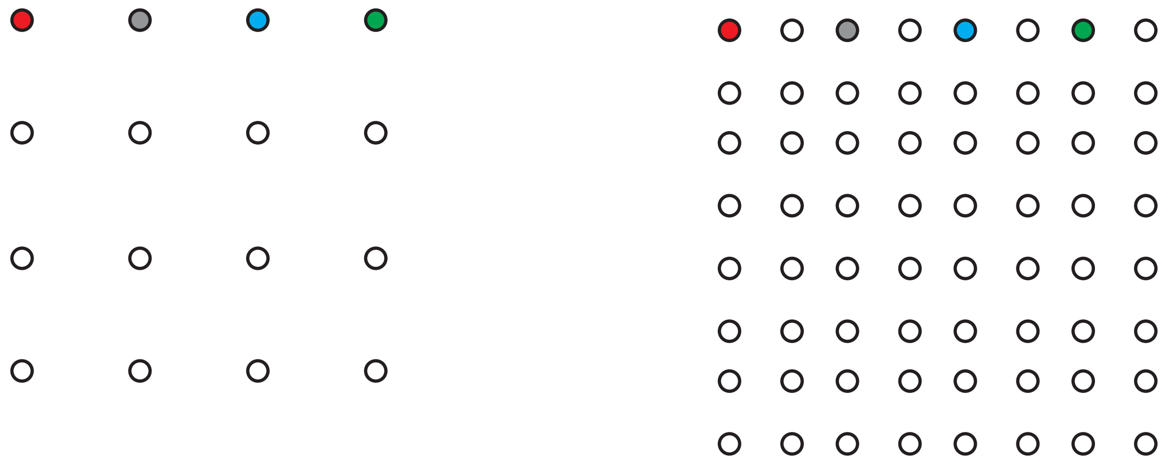
Example:

go from 4x4 to 8x8

Forward warping: bad, do not do this

scan source, place pixels in target

issue: holes in target



Upsampling

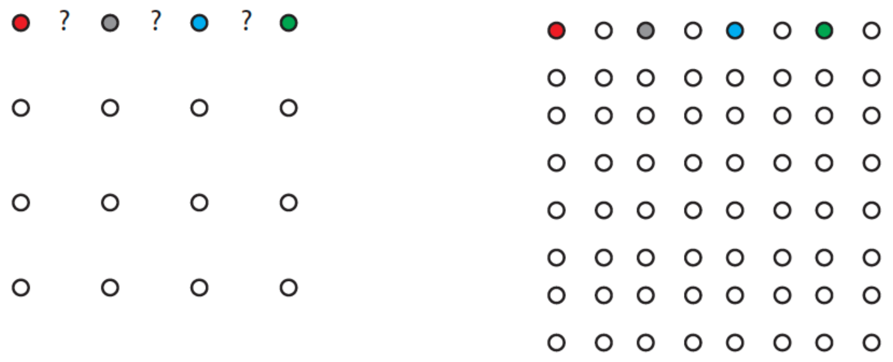
Example:

go from 4x4 to 8x8

Inverse warping:

scan target, for each pixel determine source value

usually, you must know the value at non-grid points



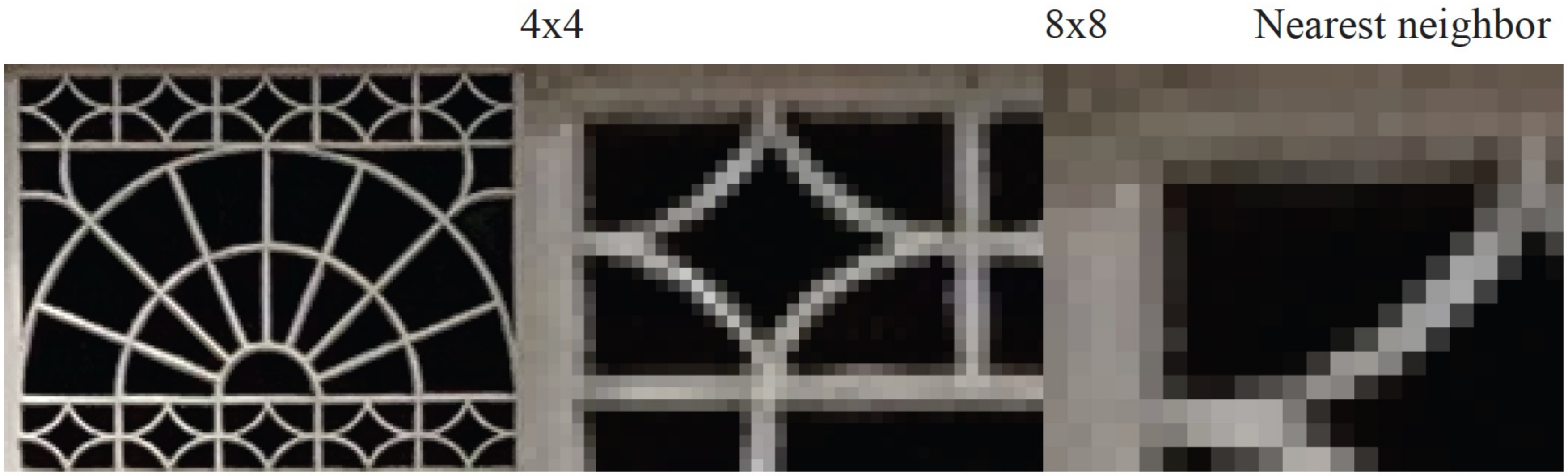
Interpolation

evaluate that function at the (likely non-integer) points. This procedure is known as *interpolation* and the function – the *interpolate* – (a) must have the same value as the original image at the original integer grid points (b) can be evaluated at any point rather than just the integer grid points. Write $\mathcal{I}(x, y)$ for an interpolate of an image $\mathcal{I}$.

Interpolation

Nearest neighbors:

The simplest interpolate is *nearest neighbors* – take the value at the integer point closest to location whose value you want. Break ties by rounding up, so you would use the value at 2, 2 if you wanted the value at 1.5, 1.5. As Figure 2.4 shows, this strategy has problems – the upsampled image looks blocky.



Interpolation

Writing nearest neighbors in a different way can be informative. For nearest neighbors, define

$$b_{nn}(u,v) = \begin{cases} 1 & \text{for } -1/2 \le u < 1/2 \text{ and } -1/2 \le u < 1/2 \\ 0 & \text{otherwise} \end{cases}$$

which has the convenient property that $b_{nn}(0,0) = 1$, but $b_{nn} = 0$ for every other set of integer coordinates. The fitted function is

$$\mathcal{I}(x,y) = \sum_{i,j} \mathcal{I}_{ij} b_{nn}(x-i, y-j).$$

and it is a simple exercise to show that it has the properties required for an interpolate. This fitted function looks like a collection of boxes, and is not continuous (Figure 2.4).

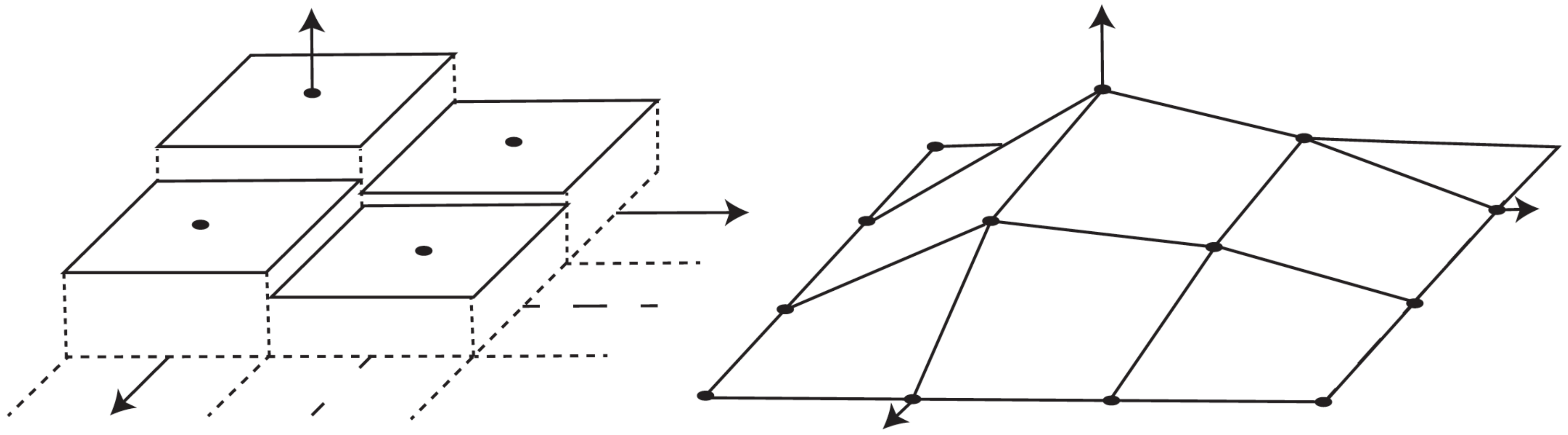


FIGURE 2.4: *On the* **left**, *a function interpolating a* 2×2 *image using nearest neighbors. The dashed lines pass through grid points, and the dotted lines are halfway between grid points. The function is zero away from the four boxes shown. Image values are shown as filled circles. On the* **right**, *a bilinear interpolate of the same data.*

Interpolation

Bilinear Interpolation

Most widely used is *bilinear interpolation*. For this, construct a function

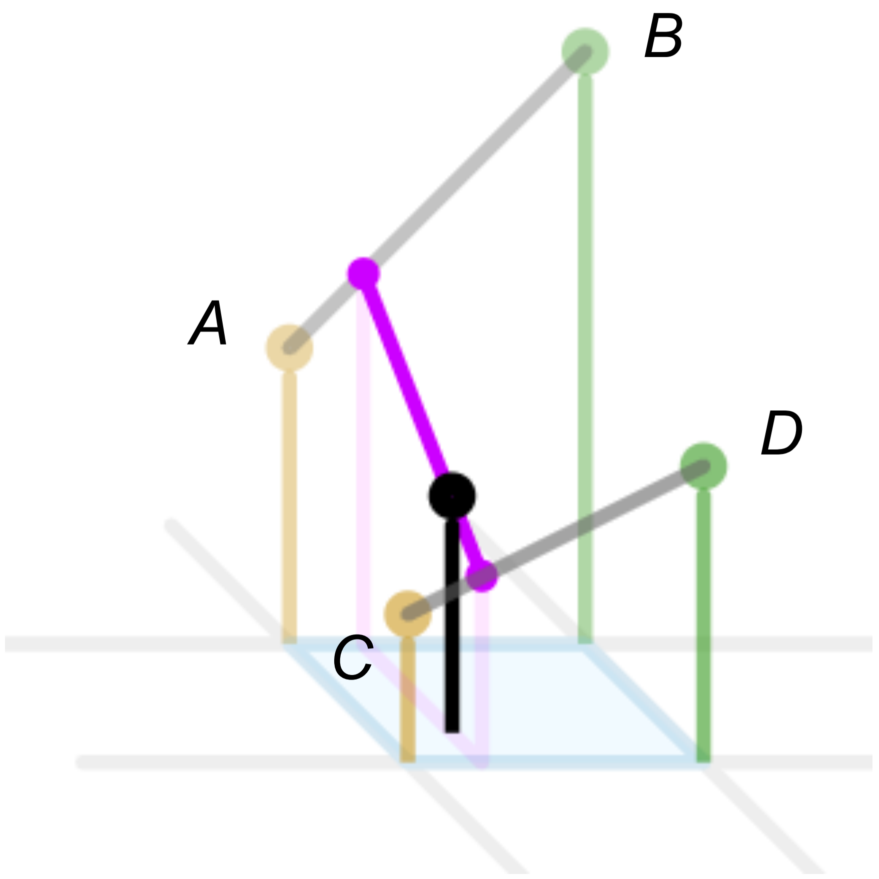
$$b_{bi}(u,v) = \begin{cases} (1-u)(1-v) & \text{for } 0 < u \leq 1 \text{ and } 0 < v \leq 1 \\ u(1-v) & \text{for } -1 \leq u \leq 0 \text{ and } 0 < v \leq 1 \\ uv & \text{for } -1 \leq u \leq 0 \text{ and } -1 \leq v \leq 0 \\ (1-u)v & \text{for } 0 < u \leq 1 \text{ and } -1 \leq v \leq 0 \\ 0 & \text{otherwise} \end{cases}$$

which is continuous, and again has the convenient property that $b_{bi}(0,0) = 1$, but $b_{nn} = 0$ for every other grid point (and looks a bit like a hat). The interpolate is

$$\mathcal{I}(x,y) = \sum_{i,j} \mathcal{I}_{ij} b_{bi}(x-i, y-j).$$

and it is a simple exercise to show that it has the properties required for an interpolate. Notice that this interpolate is continuous (Figure 2.4) and has a variety of interesting properties (**exercises**).

Bilinear interpolation more generally



http://en.wikipedia.org/wiki/Bilinear_interpolation

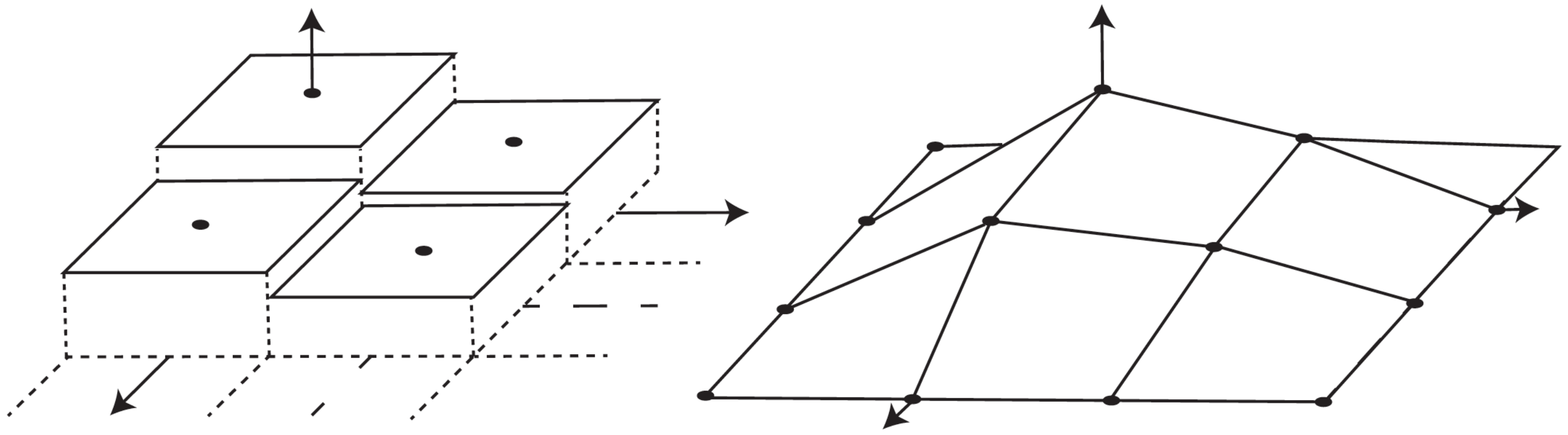
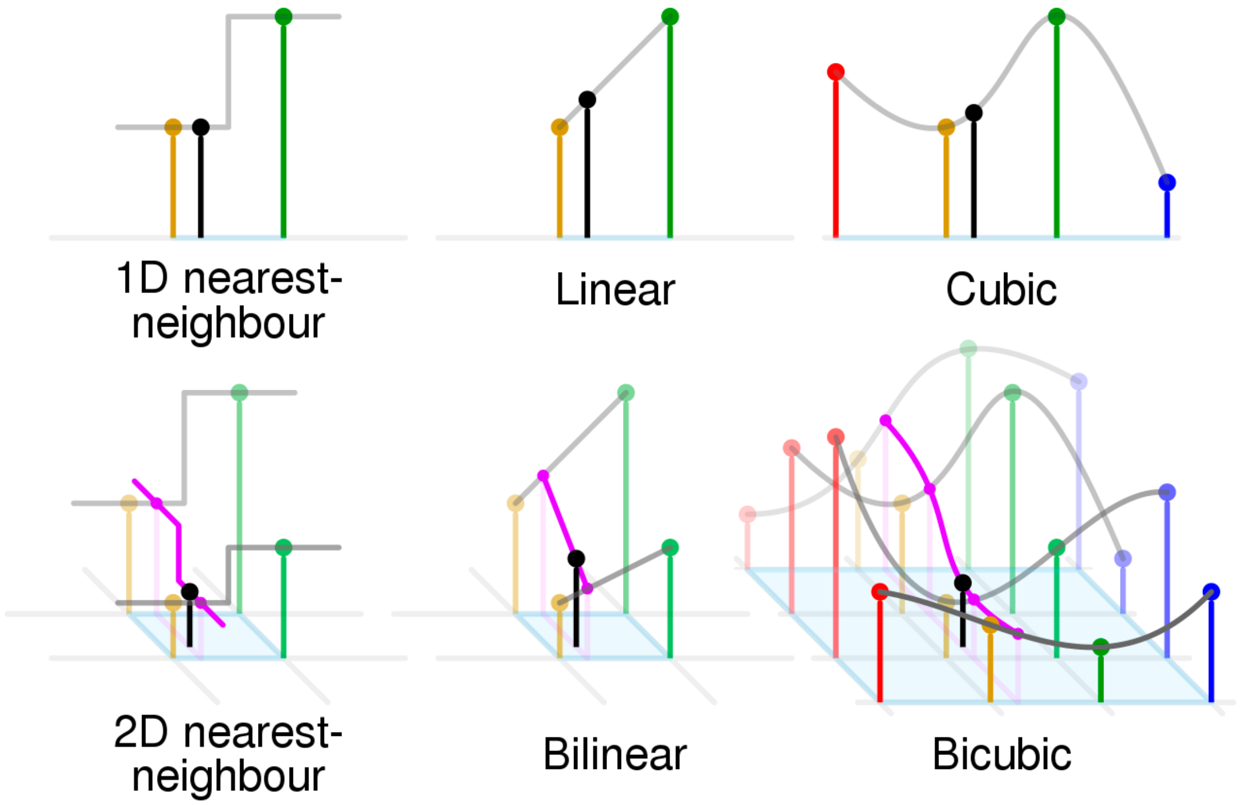
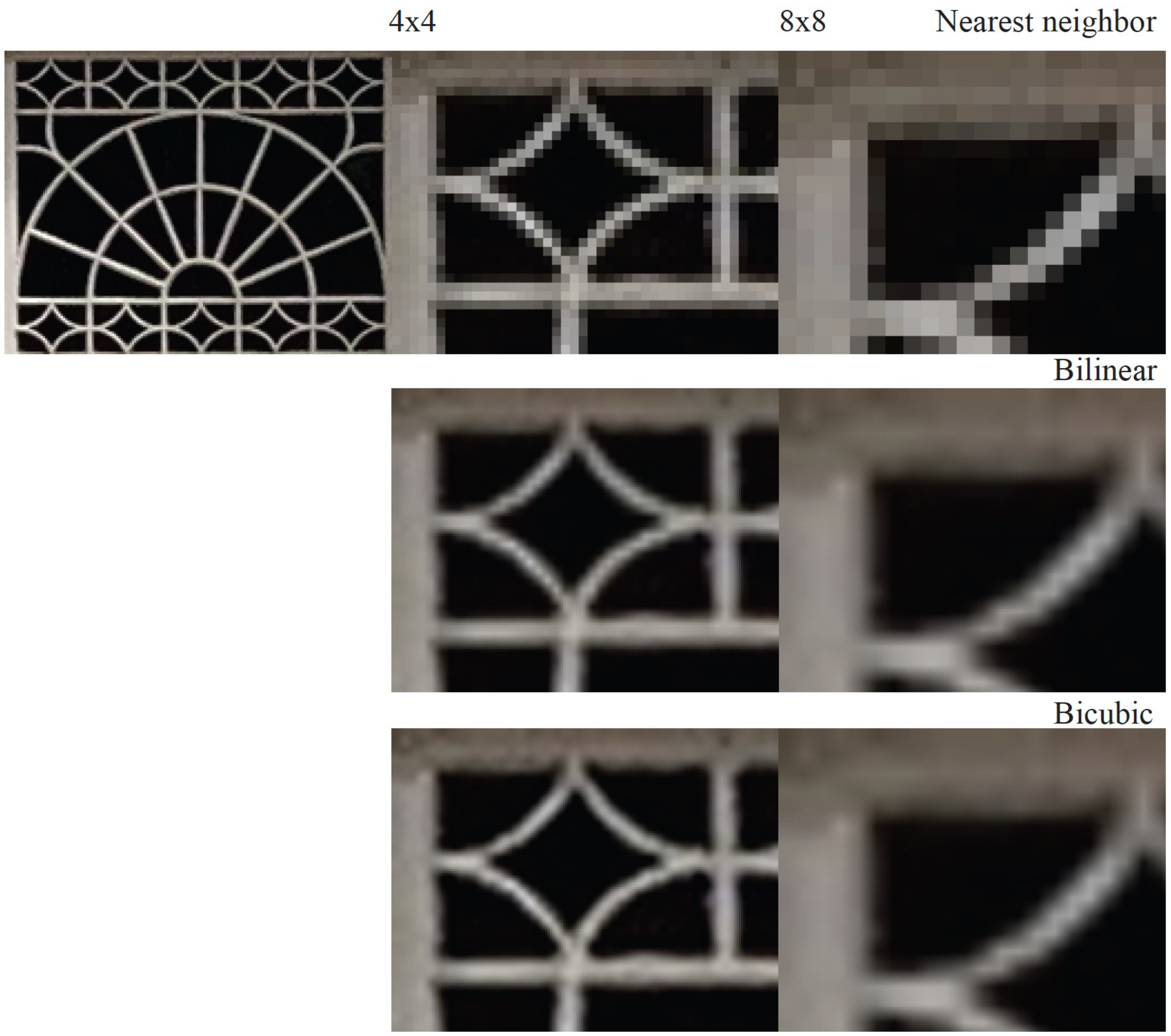


FIGURE 2.4: *On the* **left**, *a function interpolating a* 2×2 *image using nearest neighbors. The dashed lines pass through grid points, and the dotted lines are halfway between grid points. The function is zero away from the four boxes shown. Image values are shown as filled circles. On the* **right**, *a bilinear interpolate of the same data.*

Other kinds of interpolation



Source: Wikipedia


4x4
8x8
Nearest neighbor
Bilinear
Bicubic

Application: Demosaicing

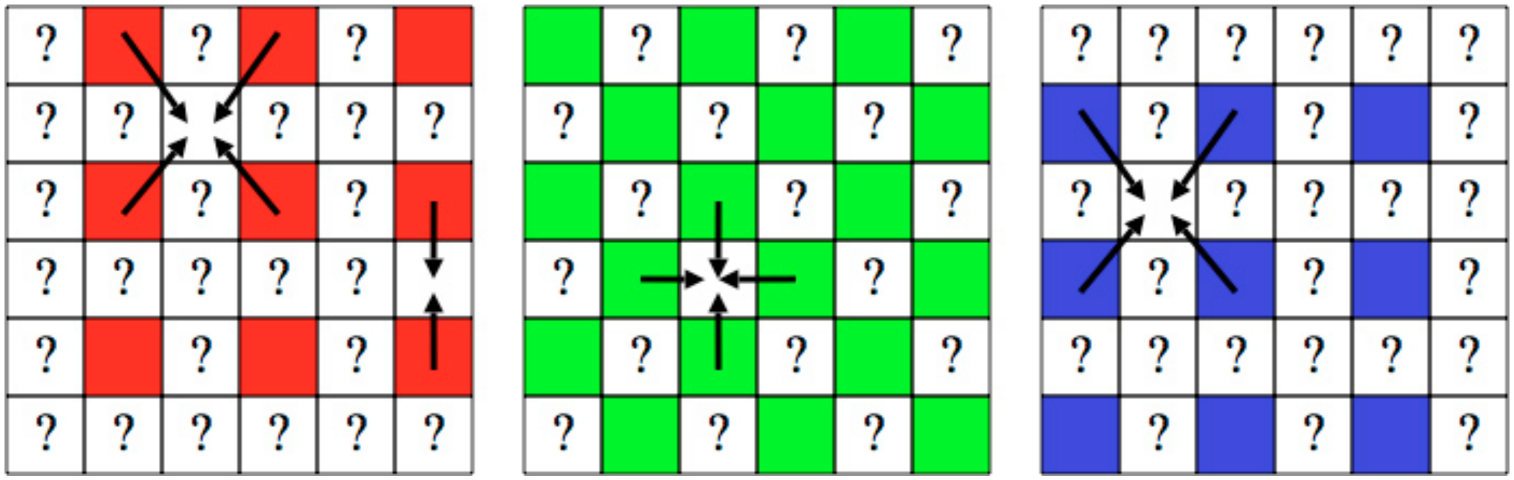
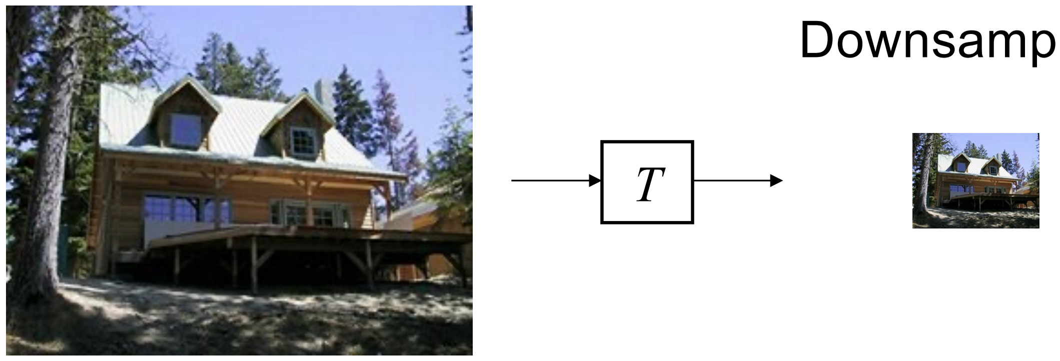
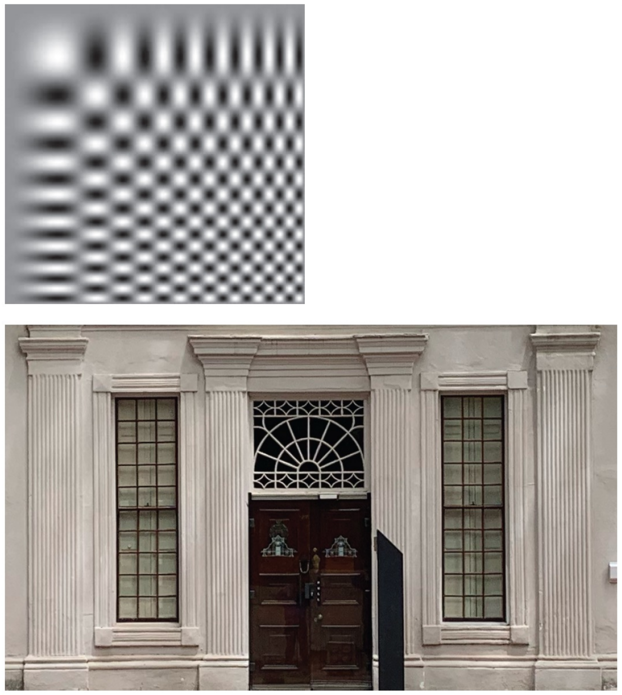


Image transformations

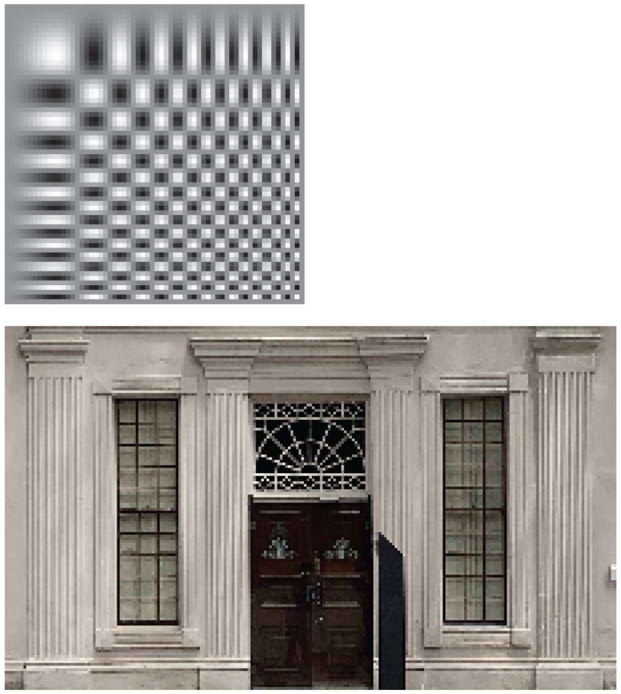
Downsampling

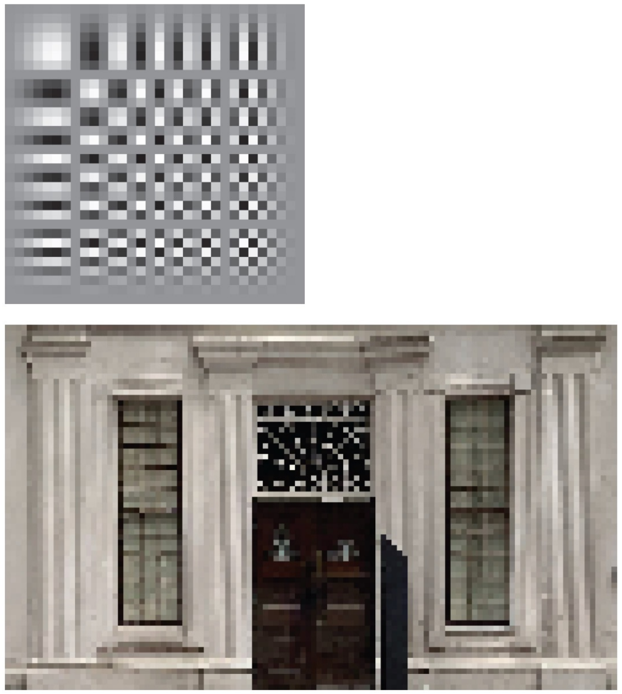


Careless downsampling creates problems



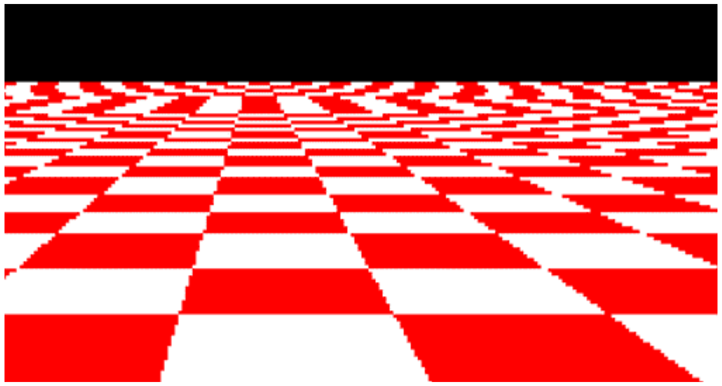
Image

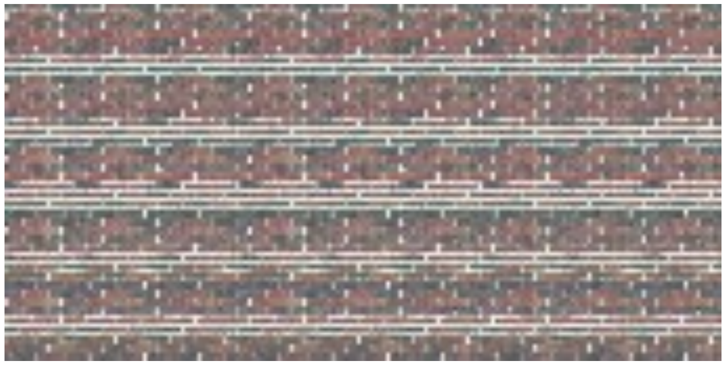

downsampled by 4


downsampled by 8

Aliasing “in the wild”

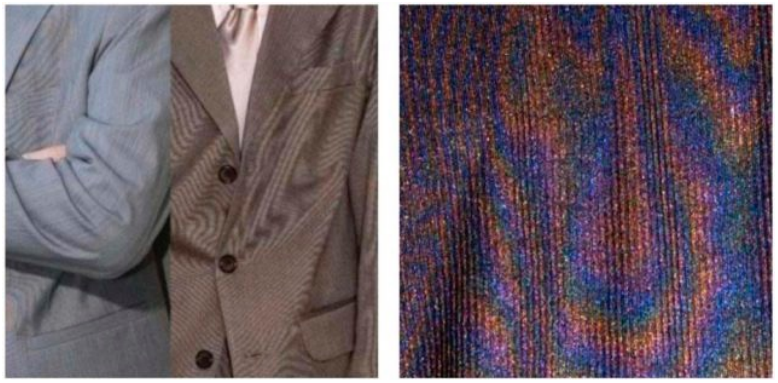
Disintegrating textures



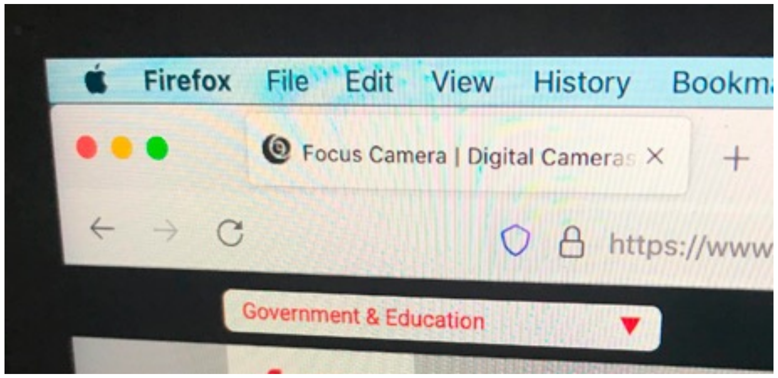


Source

Moire patterns, false color

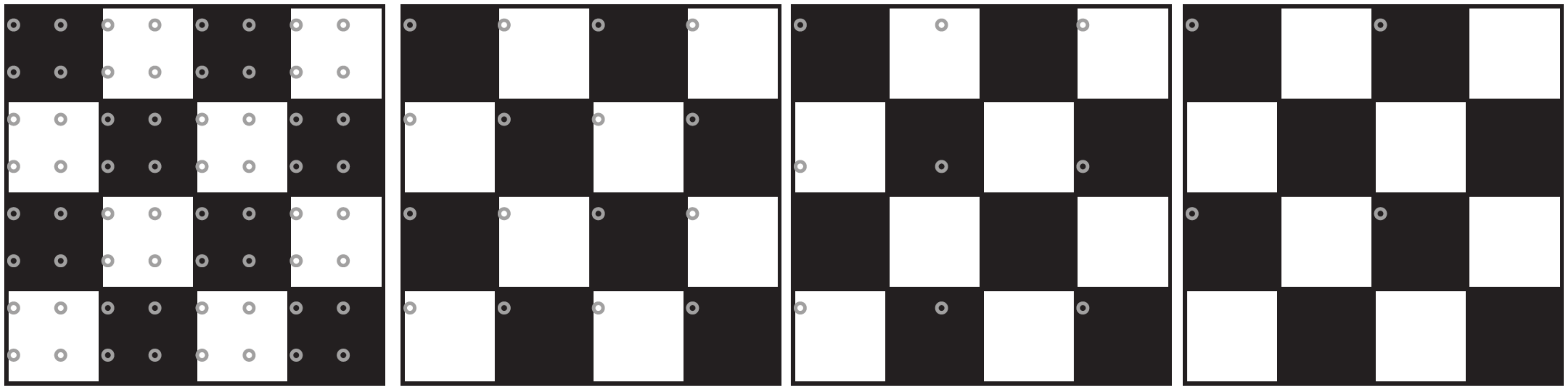


Source

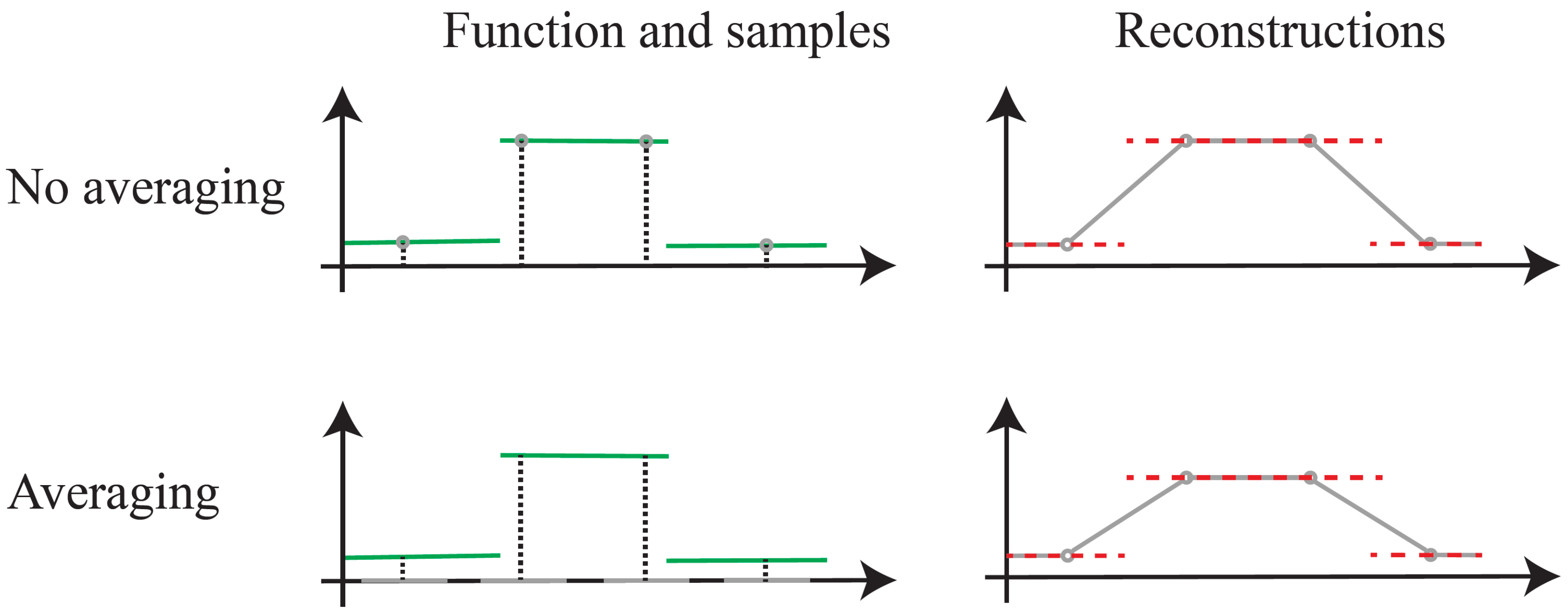


Source

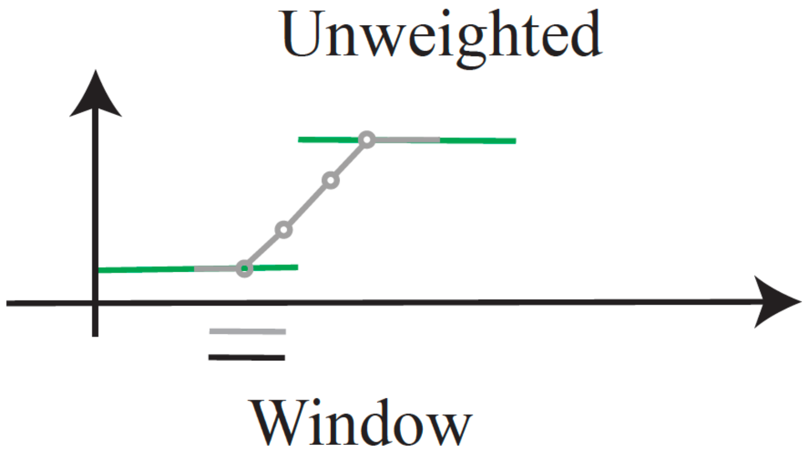
Visualizing the problems

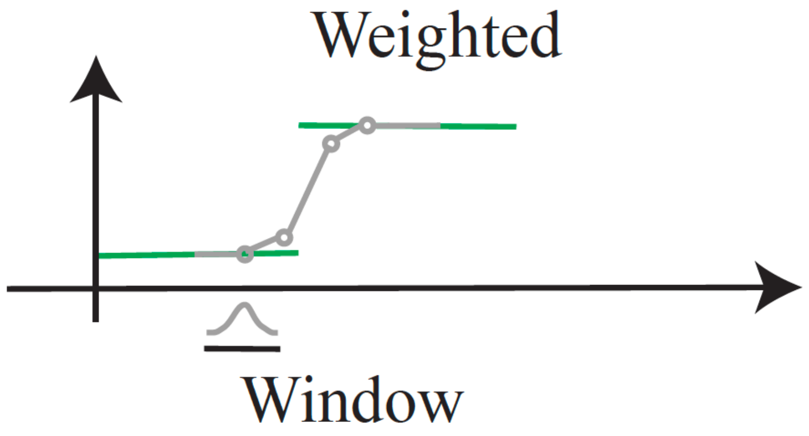


Averaging before sampling can help



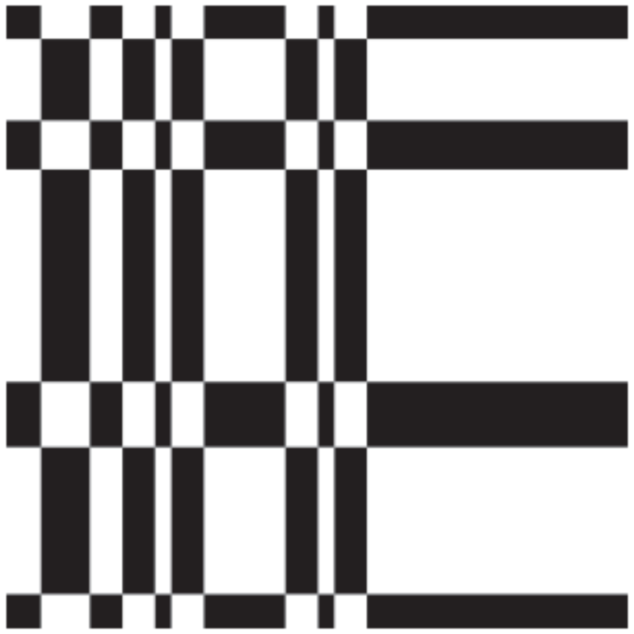
Weighting the average can help



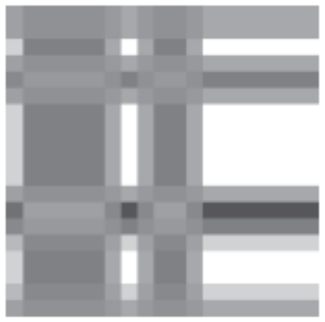


Weighting the average can help, II

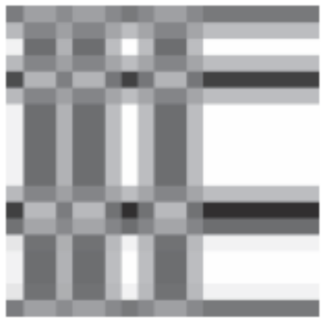
Image

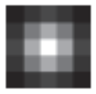


Averaged, subsampled x2



Weighted average, subsampled x2





Weights

Gaussian smoothing, I

A traditional weighting scheme is given by a one parameter family of functions, derived from the normal distribution and widely called *gaussians*. The parameter σ is sometimes called the *scale* and more usually called the *sigma* of the weights. In a $2k-1 \times 2k-1$ window, the weights will be:

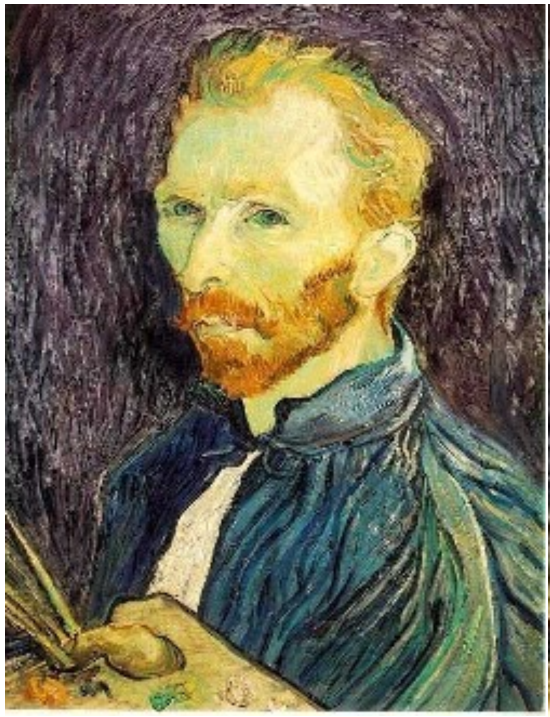
$$w_{i,j} = \frac{e - \left(\frac{(i-k)^2+(j-k)^2}{2\sigma^2}\right)}{C}$$

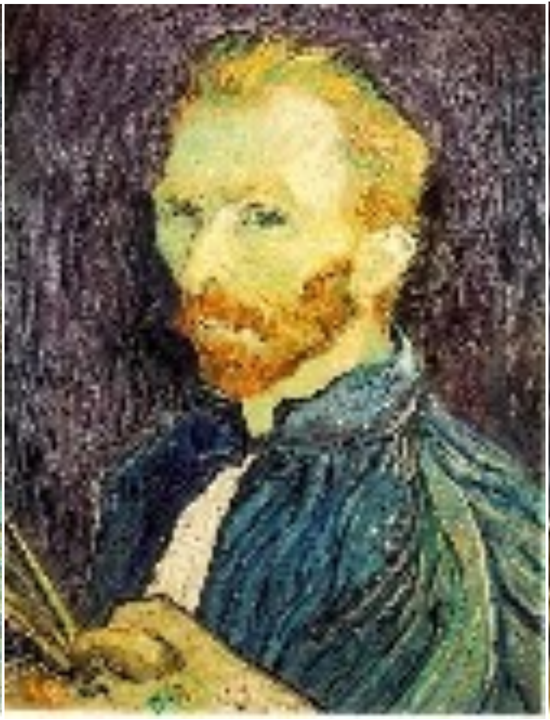
where C is chosen so the weights sum to one. Figure 2.10 shows a 5×5 window of these weights, and the considerable improvement in subsampling that can result from using a set of weights. For downsampling by a factor between one and two, $\sigma = 1$ or $\sigma = 1.5$ are fair choices.

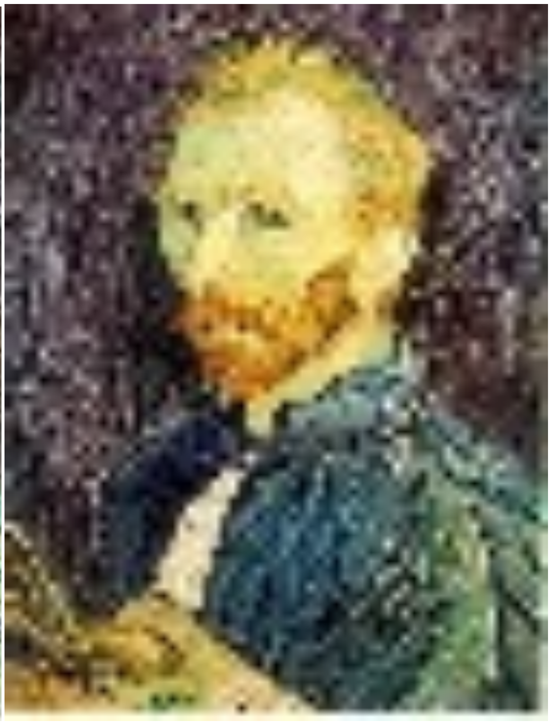
Gaussian smoothing, II

Now imagine the downsampling requires a value that *isn't* on the source grid. This value could be interpolated, but it isn't clear what to do about the smoothing. A straightforward trick applies. Take the source image $\mathcal{S}$, and form a new image $\mathcal{N}$ from that source. The i, j'th pixel in $\mathcal{N}$ is now the average of a $(2k+1) \times (2k+1)$ window of pixels in $\mathcal{S}$, centered on i, j. There are some problems when i or j are too big or too small and so the window leaves the source image. Deal with these by *padding* the source image with k rows of zeros at the top and bottom and k columns of zeros on either side. Now downsample the smoothed image $\mathcal{N}$, interpolating as required.

Subsampling without pre-filtering

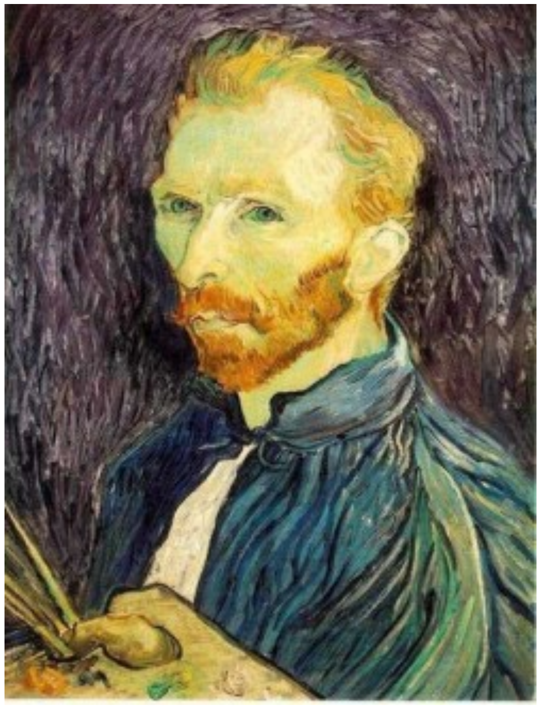

1/2

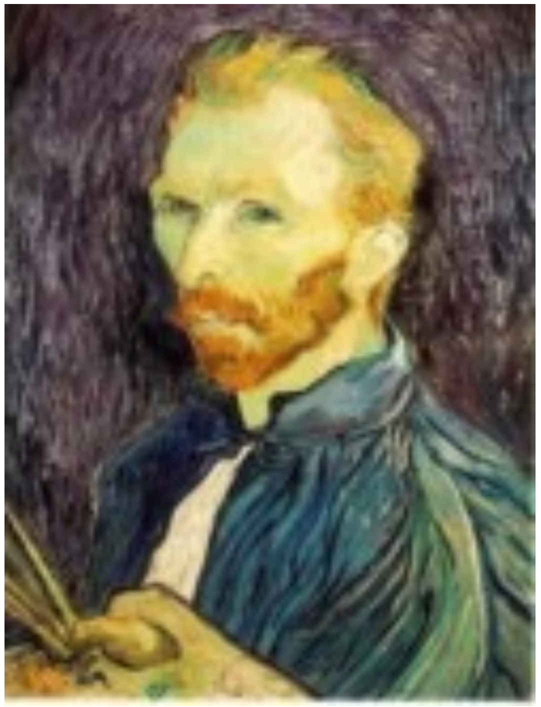

1/4 (2x zoom)

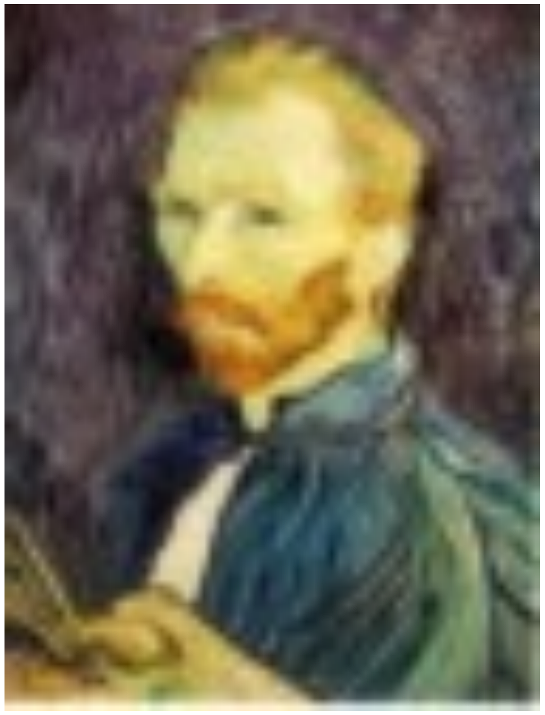

1/8 (4x zoom)

Source: S. Seitz (via D. Hoiem)

Subsampling with pre-filtering


1/2


1/4


1/8

- Image is smoothed with a *Gaussian filter* before subsampling

Source: S. Seitz (via D. Hoiem)

Gaussian pyramids

Downsample x 4

smooth with big Gaussian, downsample x 4 (NO!)

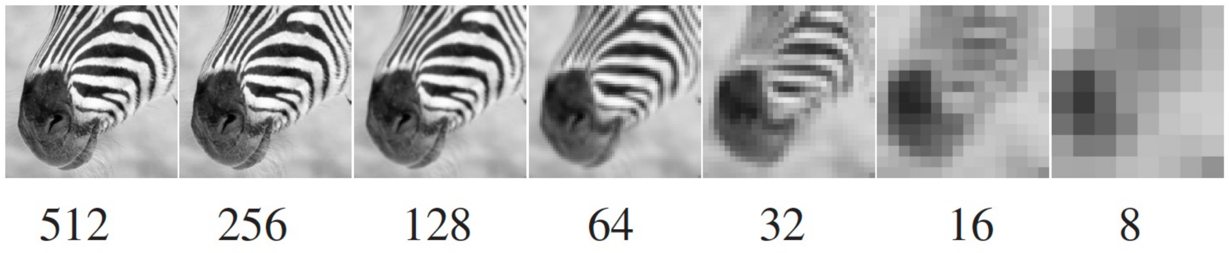
OR

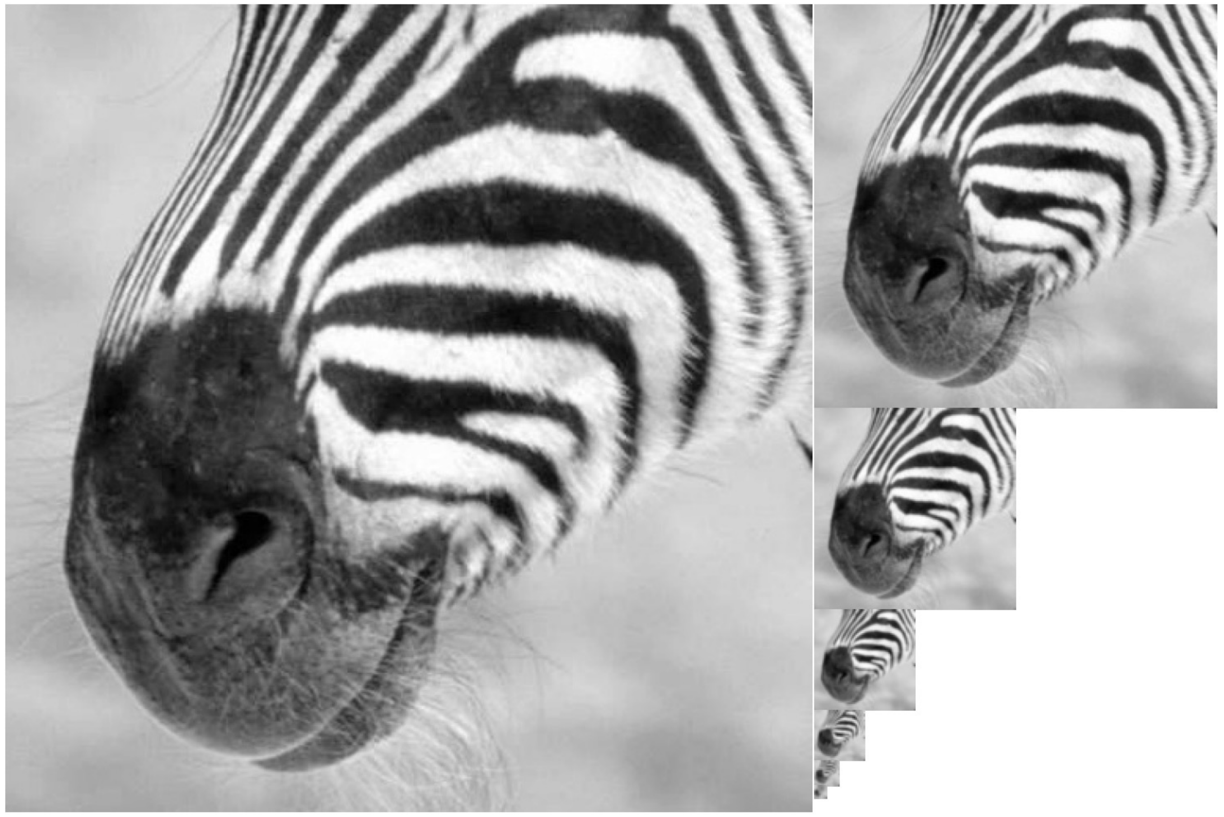
smooth, downsample x 2, smooth, downsample x 2

IDEA:

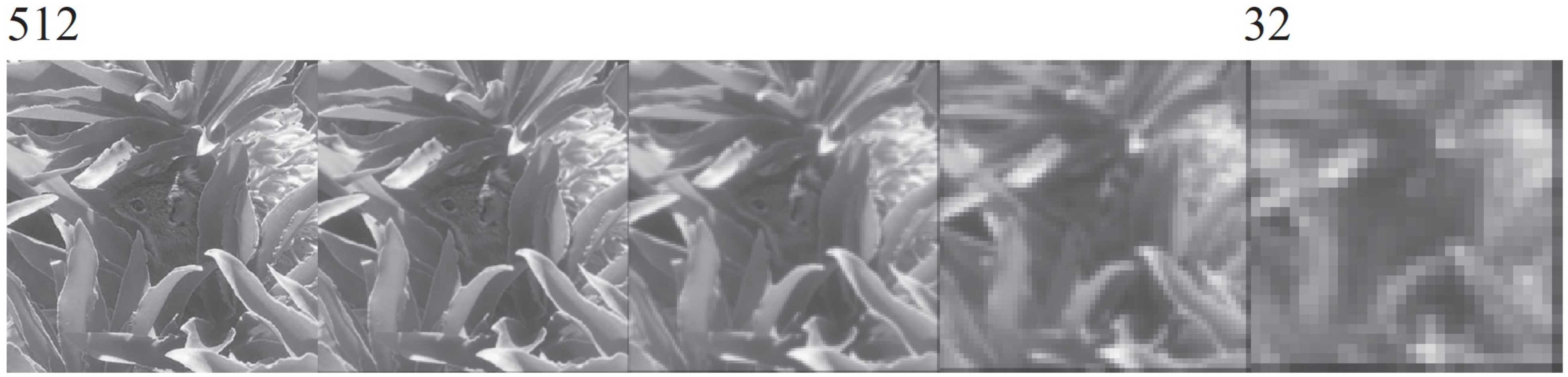
Pyramid: collection of smoothed and downsampled versions of an image

Gaussian pyramid





Gaussian pyramid



Notice that fine scale (hi res) version contains detail (mouse eye) and coarse scale (low res) shows large stuff (leaves)

Notice redundancy

coarse scale leaf predicts fine scale leaf.

Laplacian pyramid

Notation for Gaussian pyramid ->

$$\begin{aligned} G_1 &= \mathcal{I} \\ \dots \\ G_k &= D_\sigma(G_{k-1}) \\ \dots \\ G_N &= D_\sigma(G_{N-1}). \end{aligned}$$

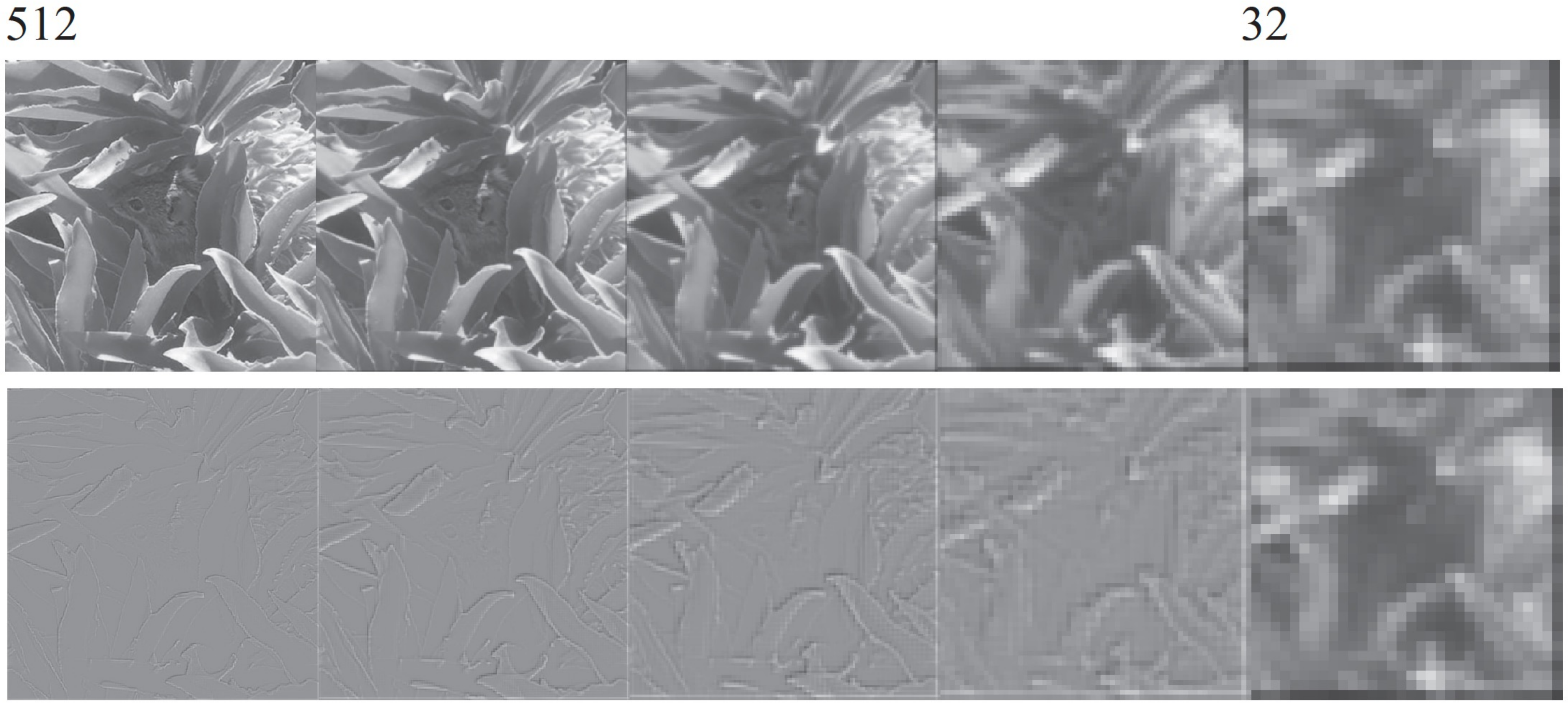
Idea:

reduce redundancy; instead of storing G_k, store G_k-Upsample(G_{k+1})

$$\begin{aligned} L_1 &= G_1 - U(D_\sigma(G_1)) \\ \dots \\ L_k &= G_k - U(D_\sigma(G_k)) \\ \dots \\ L_N &= G_N. \end{aligned}$$

Notice you can recover image
From either pyramid (Ex)

Comparing



Very simple processing with Laplacian pyramid

$$\begin{aligned} R_1 &= w(1)L_1 + R_2 \\ \ldots & \\ R_k &= w(k)L_k + R_{k+1} \ldots \\ R_N &= L_N = G_N. \end{aligned}$$

Reconstruction - all w = 1

Processing

If all the weights are 1, then $R_1 = \mathcal{I}$ (**exercises**). You can emphasize or de-emphasize some effects in the image by upweighting or downweighting the relevant scale by choosing $w(k)$. Using strongly different weights for different scales doesn't usually end well. For the example of Figure 2.13, I used weights obtained by: (a) choosing some largest scale k_x (in this case, $k_x = 3$); (b) choosing a weight α then (c) forming

$$w(k) = \left(1 + \left[\alpha \frac{\max k_x - k0}{k_x}\right]\right).$$

Figure 2.13 shows how various choices of α either sharpen or smooth the image.

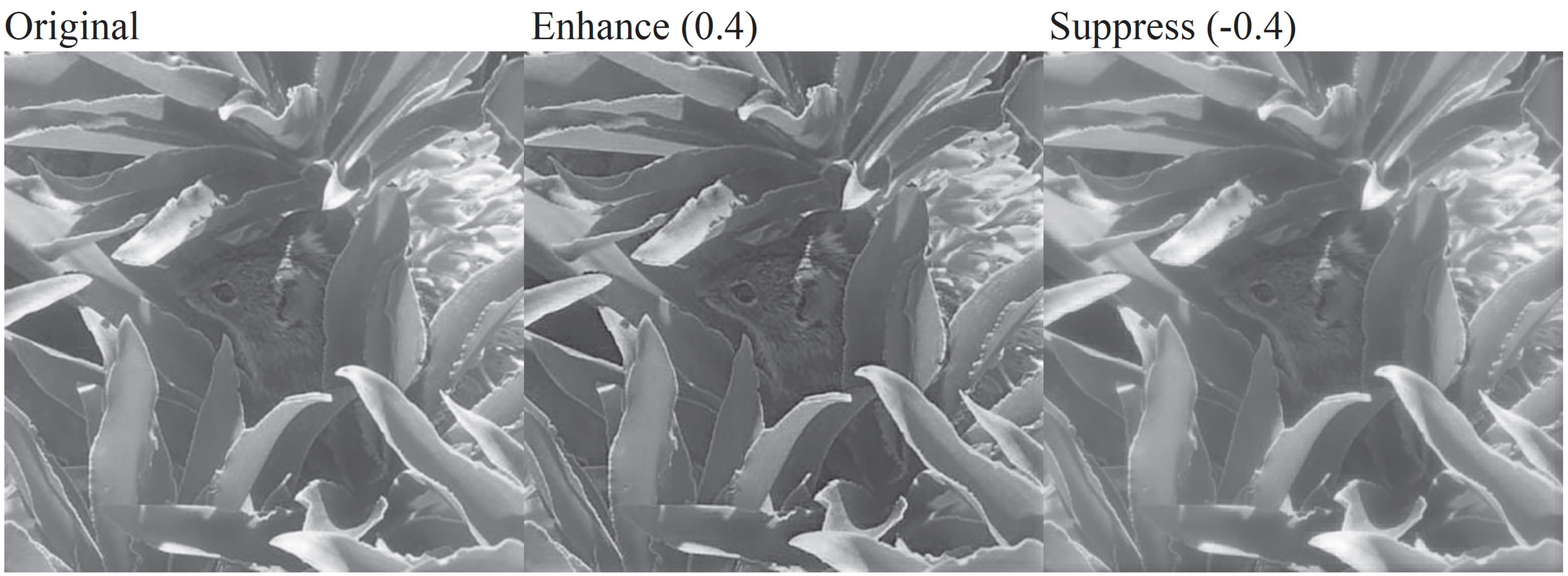


FIGURE 2.13: *Images can be reconstructed from Laplacian pyramids, and weighting components can emphasize or smooth edges. The Laplacian pyramid of Figure 2.12, reconstructed into an image using the method of Section 2.3.5, with* $\alpha = 0$ *(***left***; original image);* $\alpha = 0.4$ *(***center***; emphasizes edges); and* $\alpha = -0.4$ *(***right***; smoothes edges).*