Lecture 2:

The Camera Image Processing Pipeline

Visual Computing Systems
Stanford CS348K, Fall 2018

Acknowledgement to Ren Ng (Berkeley), Marc Levoy (Stanford/Google) for various slides used in this lecture.

The next two lectures…

The pixels you see on screen are quite different than the values recorded by the sensor in a modern digital camera.

Computation is now a fundamental aspect of producing high-quality pictures.

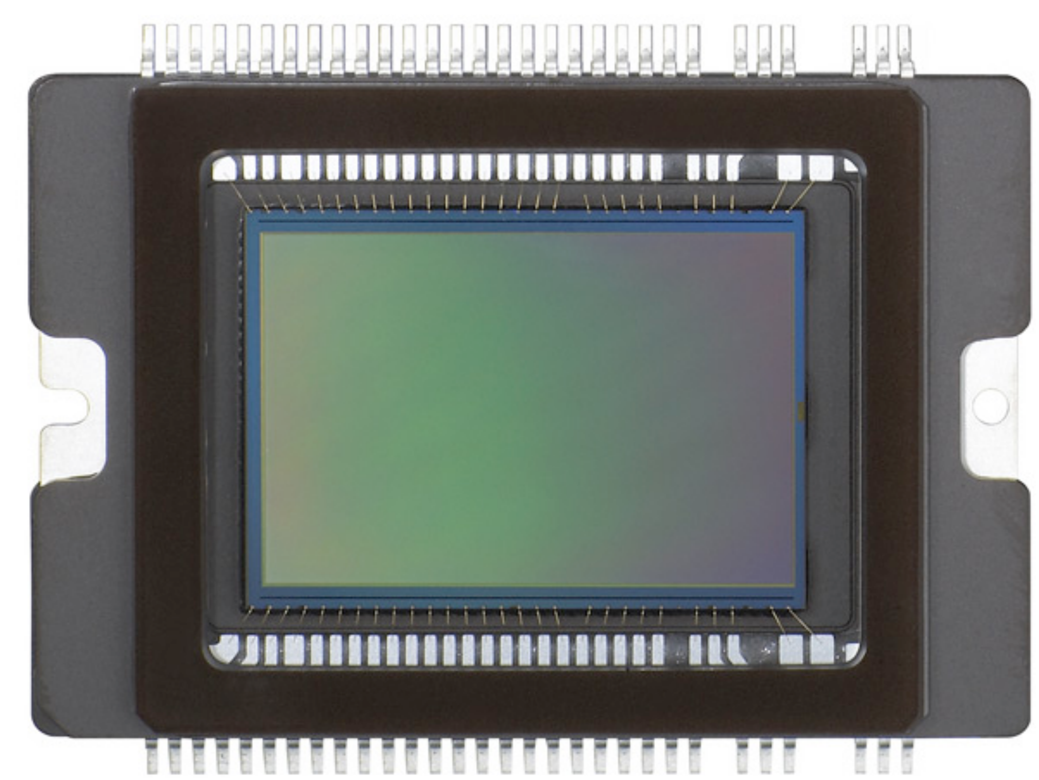
Sensor output ("RAW") → **Computation** →

Beautiful image that impresses your friends on Instagram

Part 1: image sensing hardware

(how a digital camera measures light, and how physical limitations of these devices place challenges on software)

Camera cross section

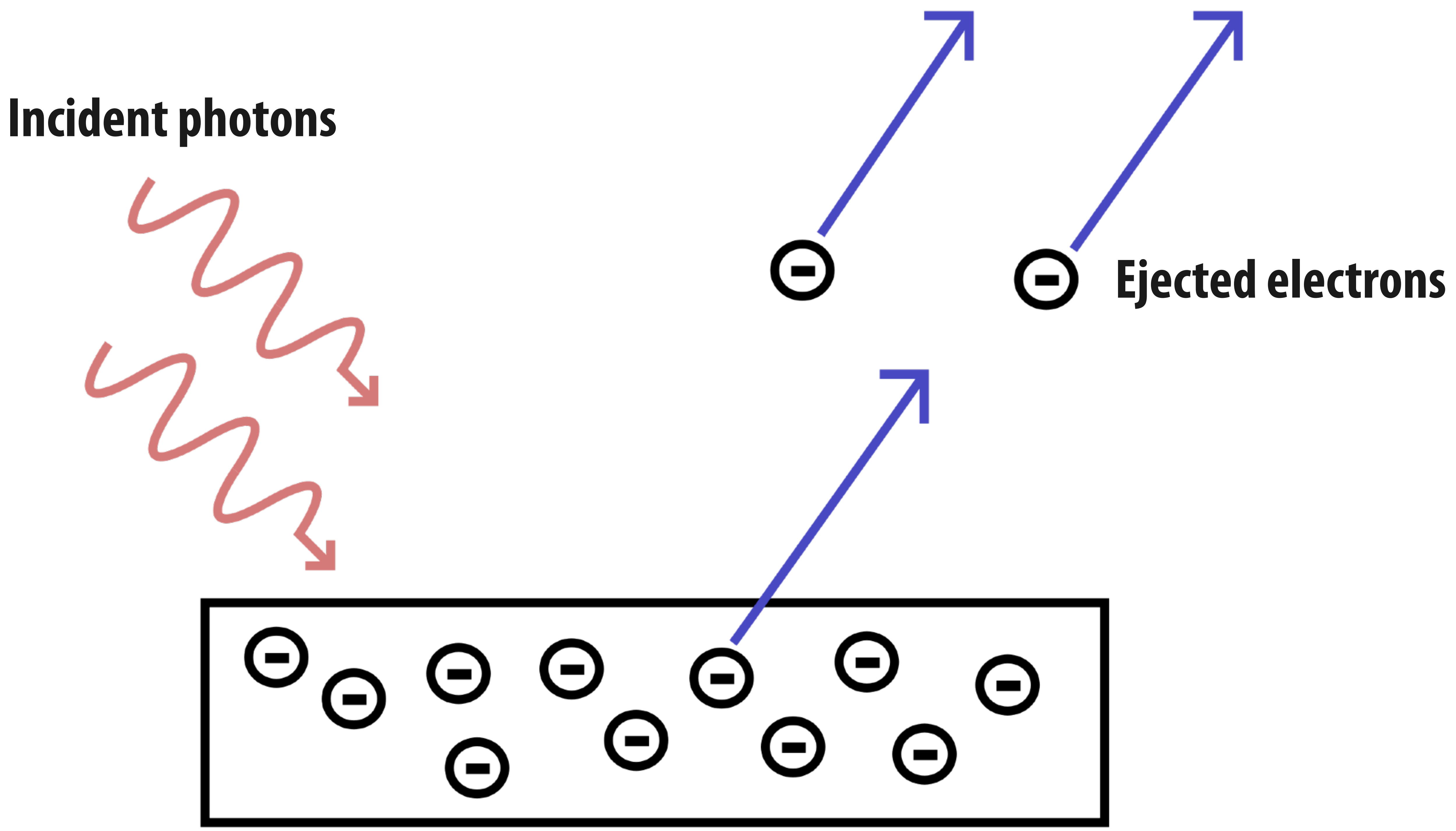


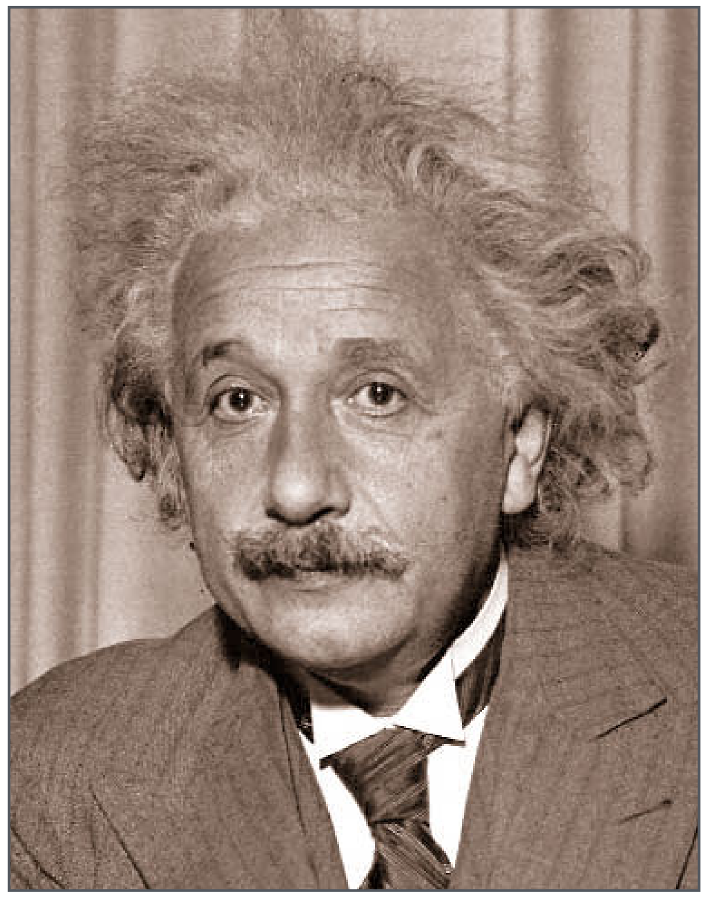
Canon 14 MP CMOS Sensor
(14 bits per pixel)

Image credit: Canon (EOS M)

The Sensor

Photoelectric effect



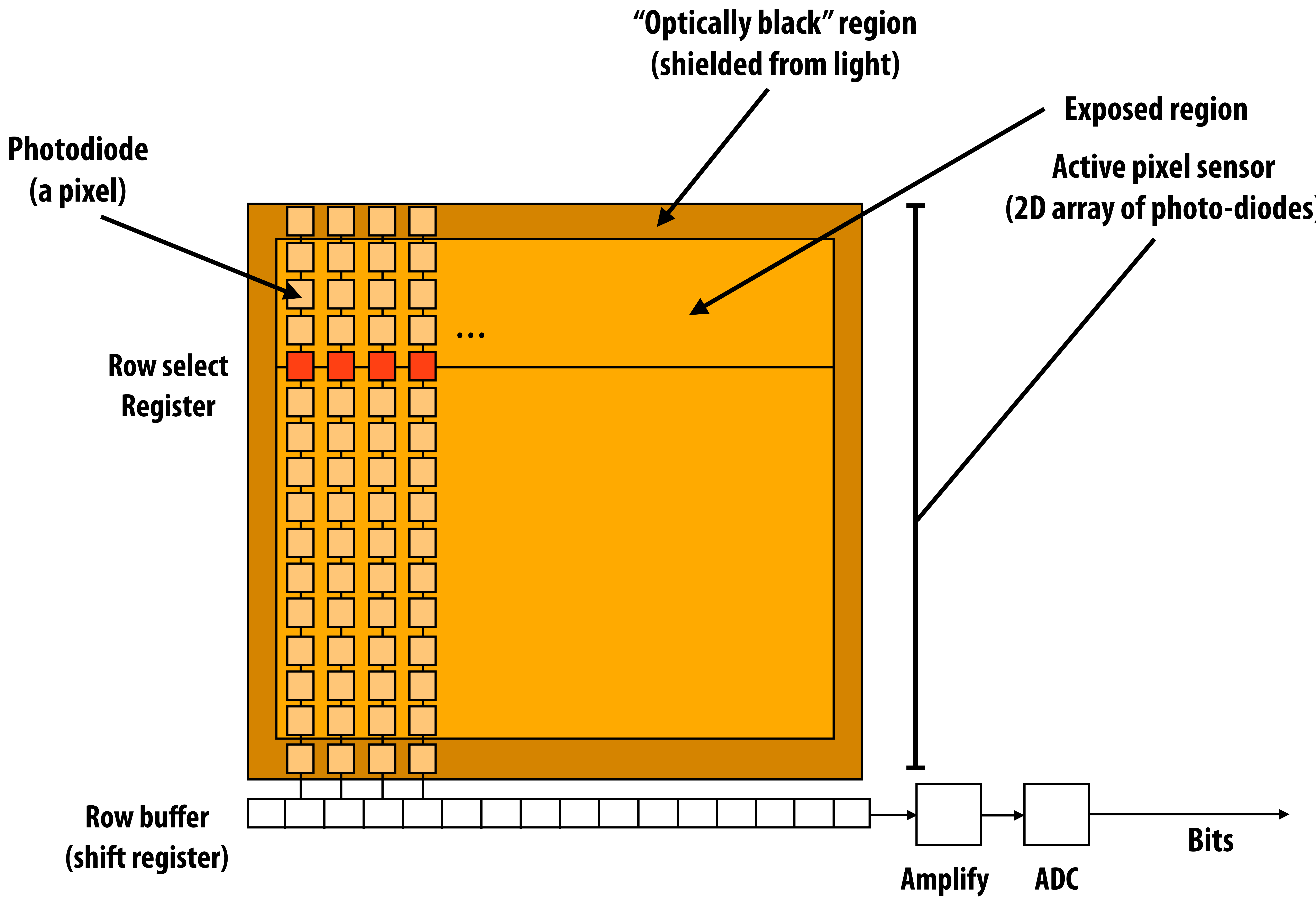


Albert Einstein

Einstein's Nobel Prize in 1921 "for his services to Theoretical Physics, and especially for his discovery of the law of the photoelectric effect"

Slide credit: Ren Ng

CMOS sensor



"Optically black" region
(shielded from light)
Exposed region
Active pixel sensor
(2D array of photo-diodes)
Photodiode
(a pixel)
...
Row select
Register
Row buffer
(shift register)
Amplify
ADC
Bits

CMOS APS (active pixel sensor) pixel

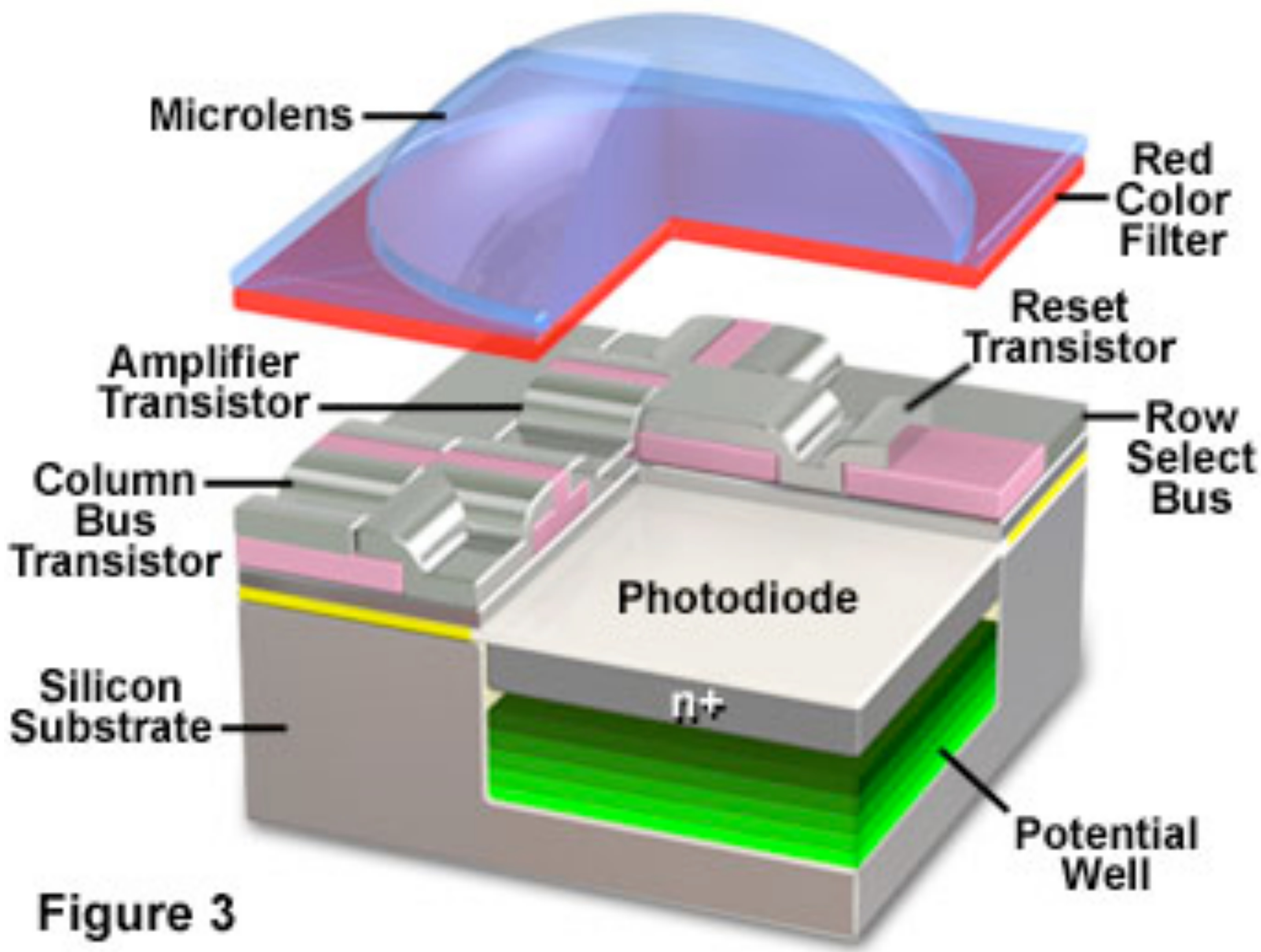


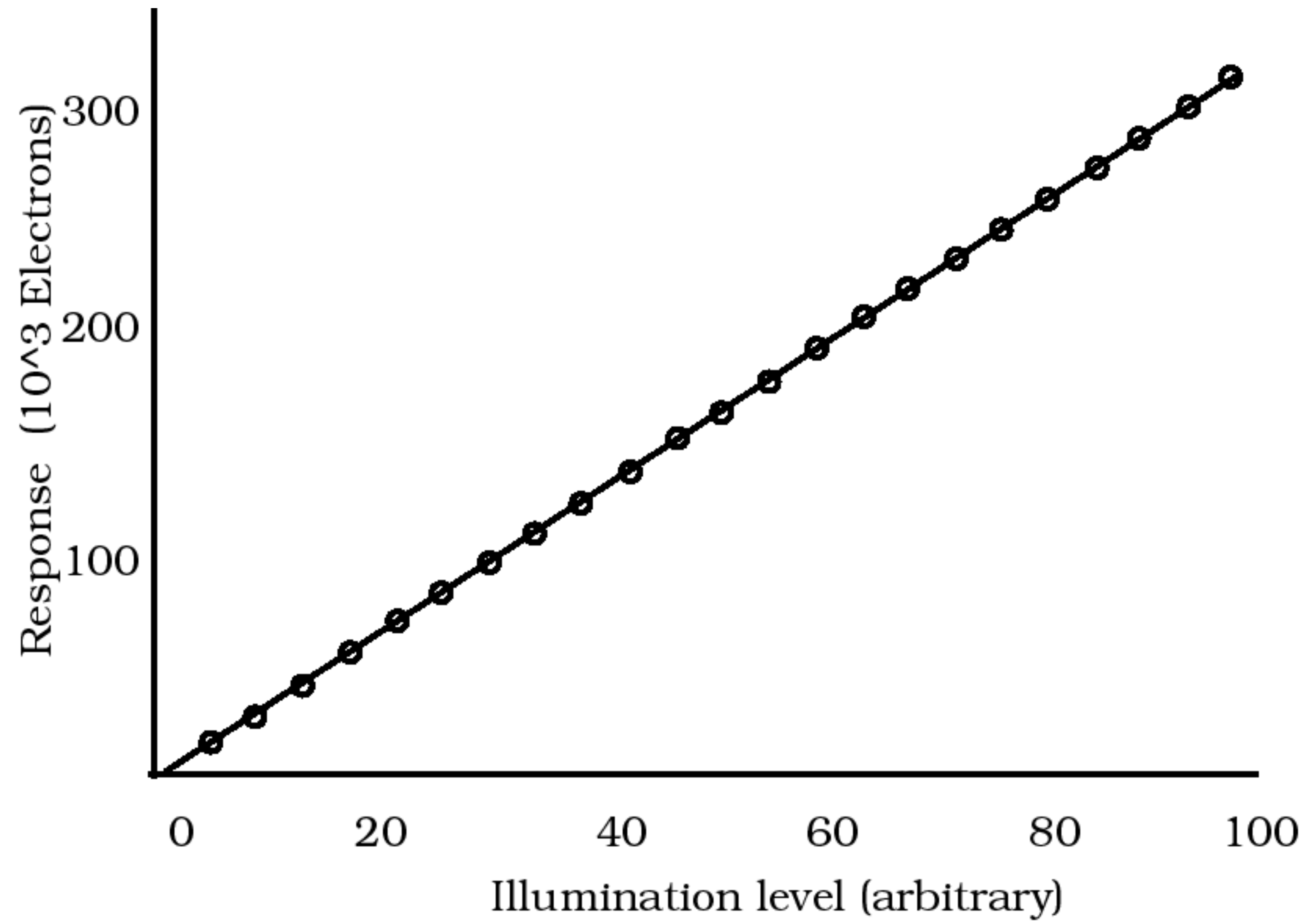
Illustration credit: Molecular Expressions (http://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html)

CMOS response functions are linear

Photoelectric effect in silicon:

- **Response function from photons to electrons is linear**

  (Some nonlinearity close to 0 due to noise and when close to pixel saturation)



(Epperson, P.M. et al. Electro-optical characterization of the Tektronix TK5 ..., Opt Eng., 25, 1987)

Slide credit: Ren Ng

Quantum efficiency

- Not all photons will produce an electron
  - Depends on quantum efficiency of the device

$$QE = \frac{\#\, electrons}{\#\, photons}$$

  - Human vision: ~15%
  - Typical digital camera: < 50%
  - Best back-thinned CCD: > 90% (e.g., telescope)

Slide credit: Ren Ng

Sensing Color

Electromagnetic spectrum

Describes distribution of power (energy/time) by wavelength

Below: spectrum of various common light sources:

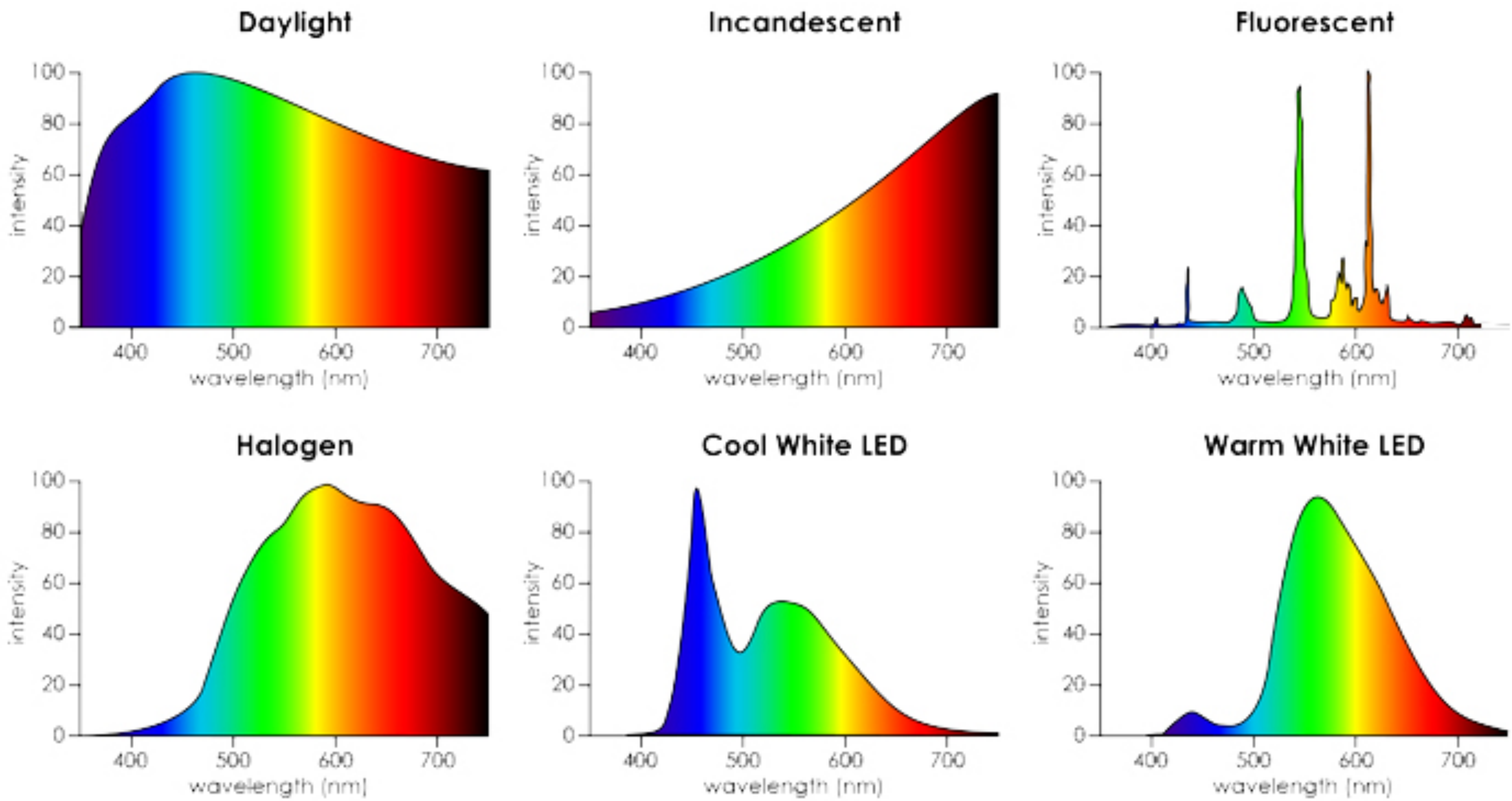


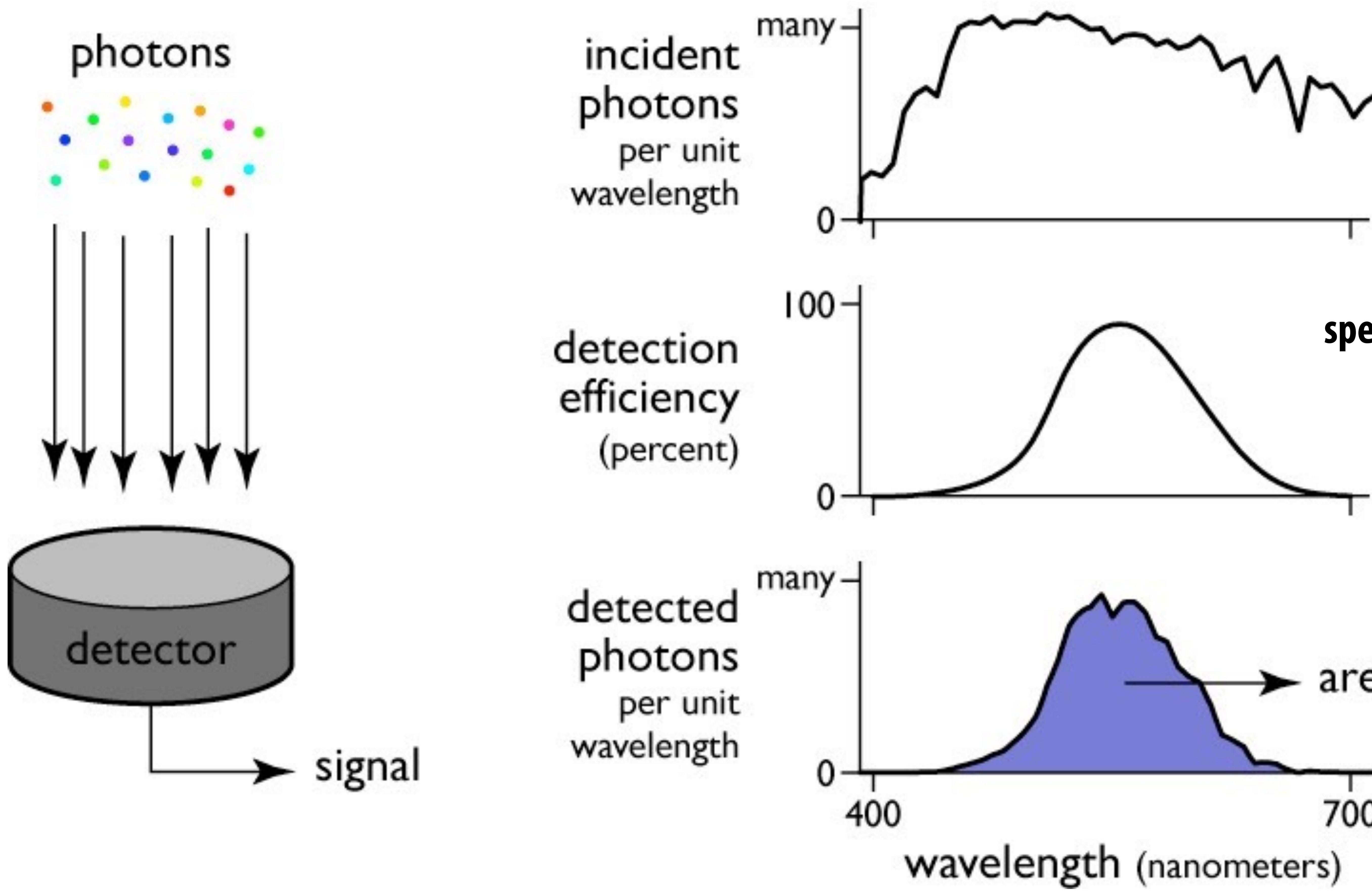
Figure credit:

admesy
ADVANCED MEASUREMENT SYSTEMS

Example: warm white vs. cool white

Image credit: (Oz Lighting: https://www.ozlighting.com.au/blog/what-is-warm-white-versus-cool-white/)

Simple model of a light detector



incoming spectrum

$\Phi(\lambda)$

spectral response function

$r(\lambda)$

(overall response)

$$R = \int_\lambda \Phi(\lambda) r(\lambda) d\lambda$$

Figure credit: Steve Marschner

Spectral response of cone cells in human eye

Three types of cells in eye responsible for color perception: S, M, and L cones (corresponding to peak response at short, medium, and long wavelengths)

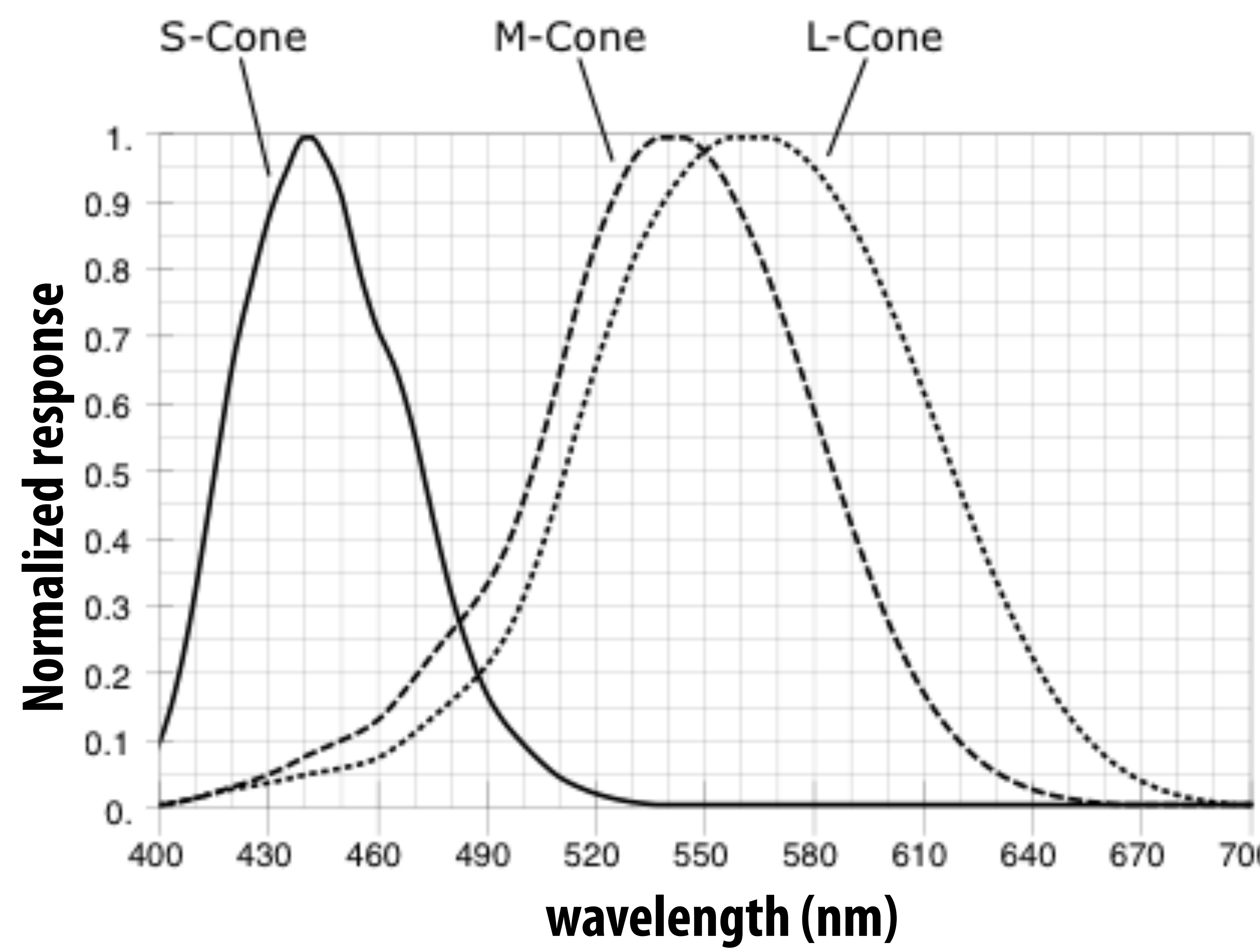
Implication: the space of human-perceivable colors is three dimensional

$$S = \int_\lambda \Phi(\lambda) S(\lambda) d\lambda$$

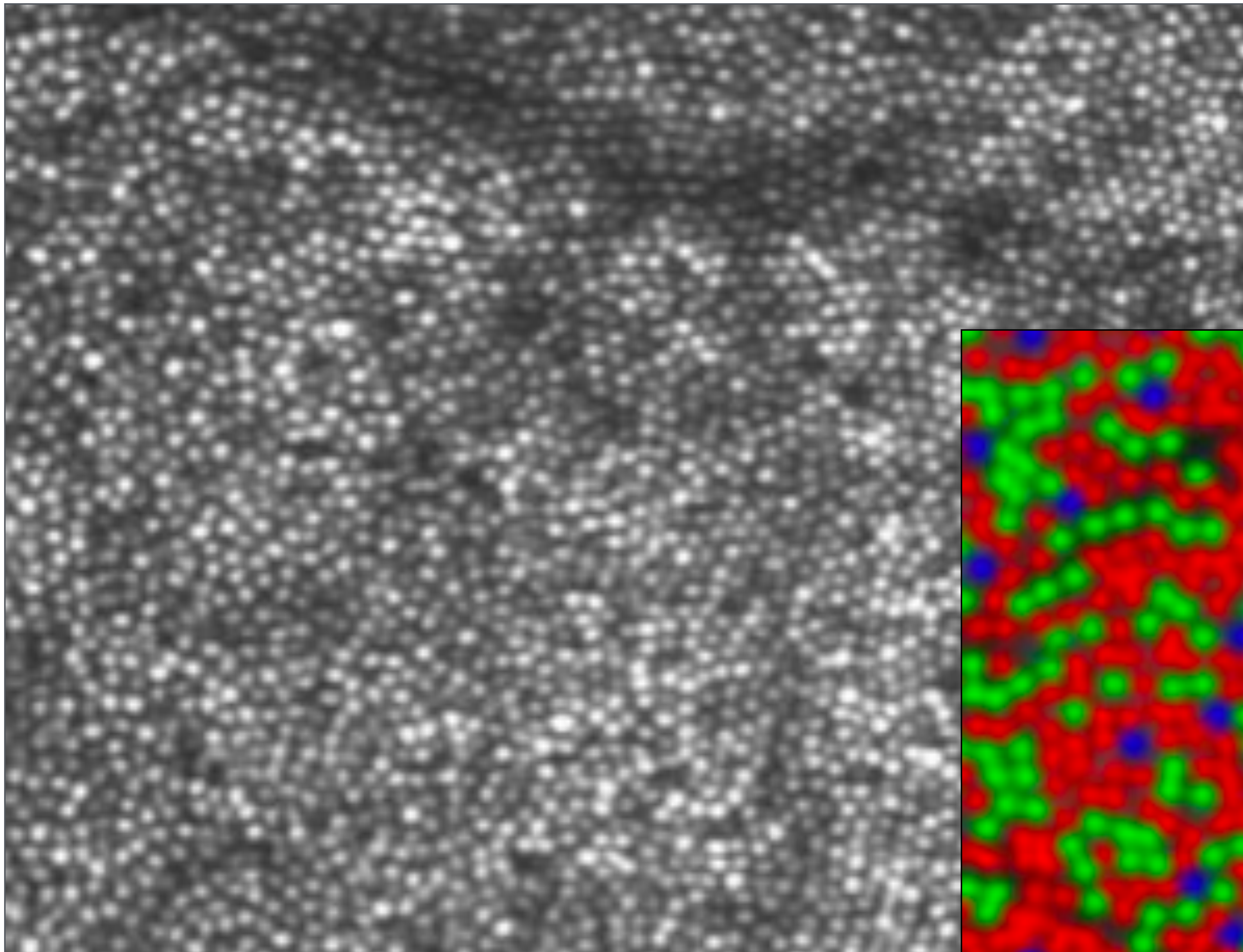
$$M = \int_\lambda \Phi(\lambda) M(\lambda) d\lambda$$

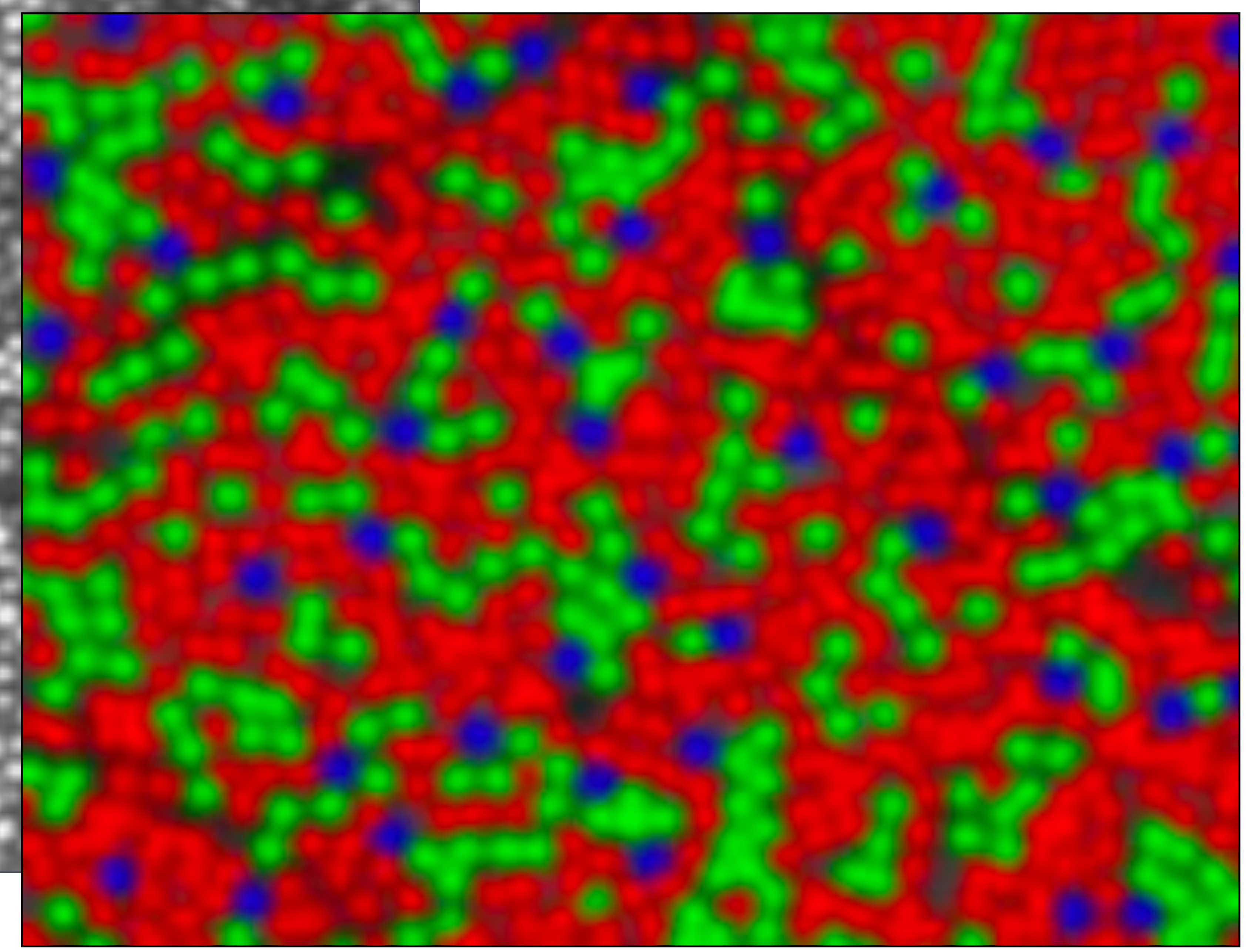
$$L = \int_\lambda \Phi(\lambda) L(\lambda) d\lambda$$

Response functions for S, M, and L cones



Human eye cone cell mosaic



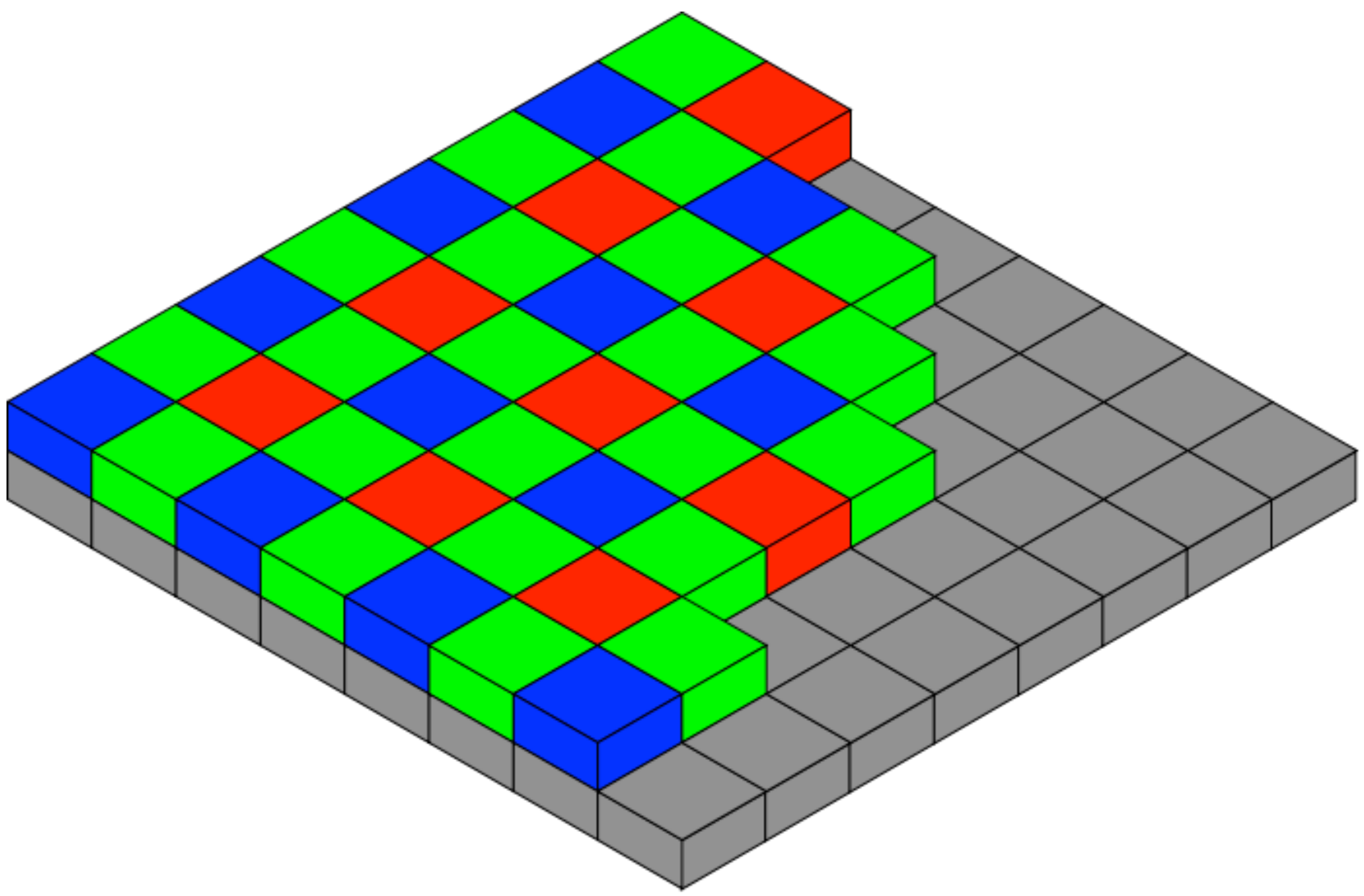


False color image:
red = L cones
green = M cones
blue = R cones

Image Credit: Ramkumar Sabesan Lab

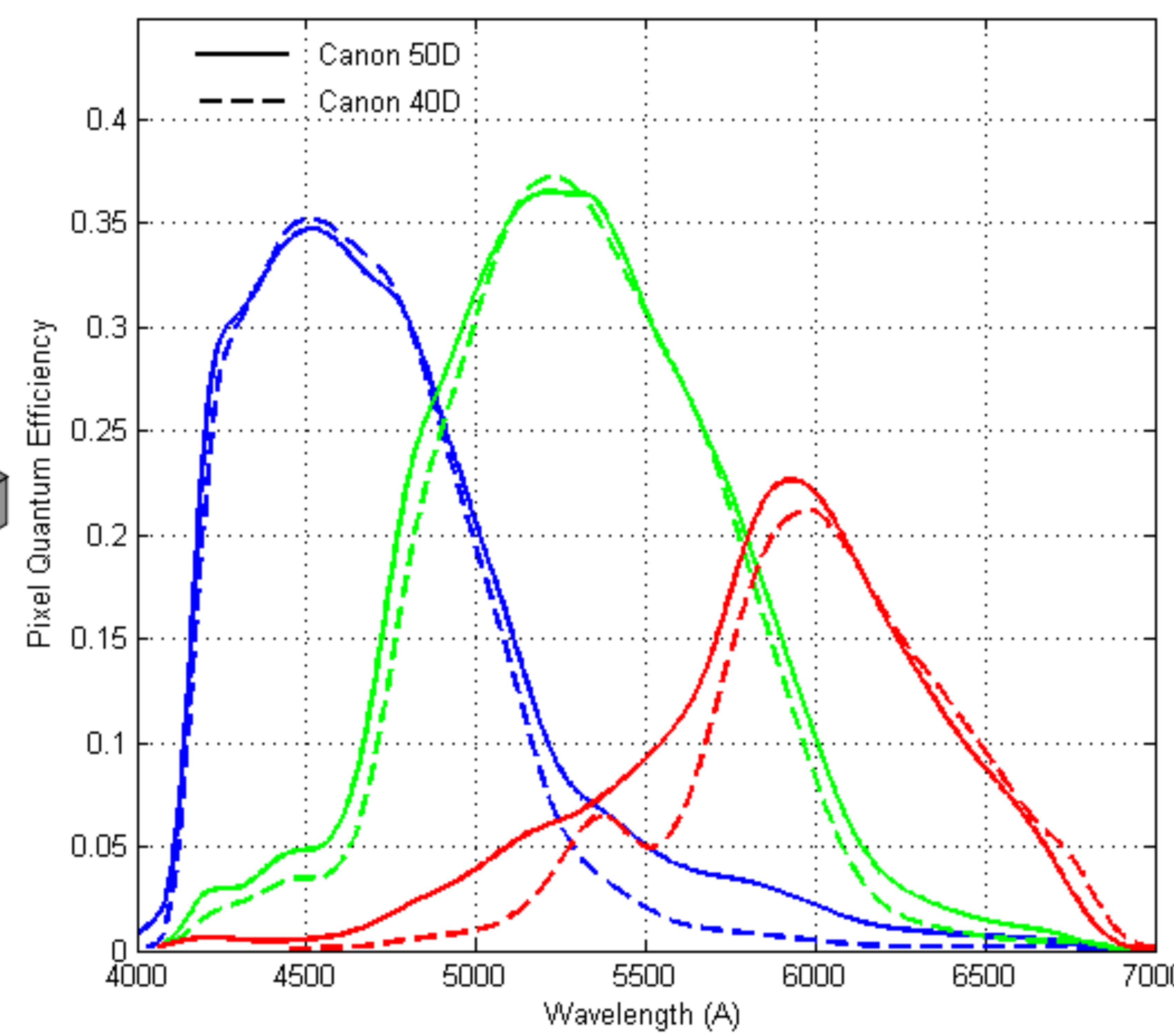
Color filter array (Bayer mosaic)

- Color filter array placed over sensor
- Result: different pixels have different spectral response (each pixel measures red, green, or blue light)
- 50% of pixels are green pixels



Traditional Bayer mosaic
(other filter patterns exist: e.g., Sony's RGBE)

Pixel response curve: Canon 40D/50D



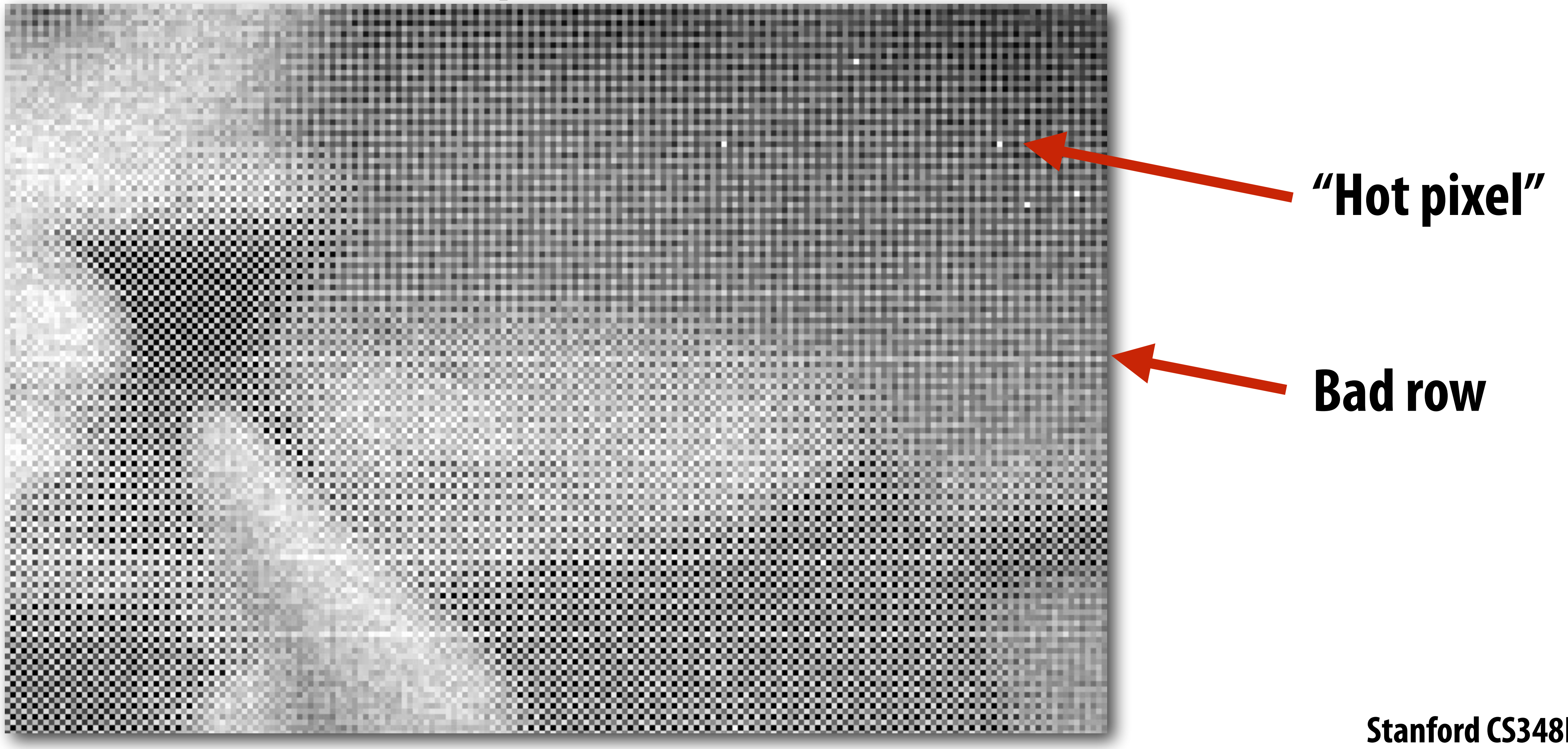
$$f(\lambda)$$

Image credit:
Wikipedia, Christian Buil (http://www.astrosurf.com/~buil/cameras.htm)

RAW sensor output (simulated data)

Light hitting sensor

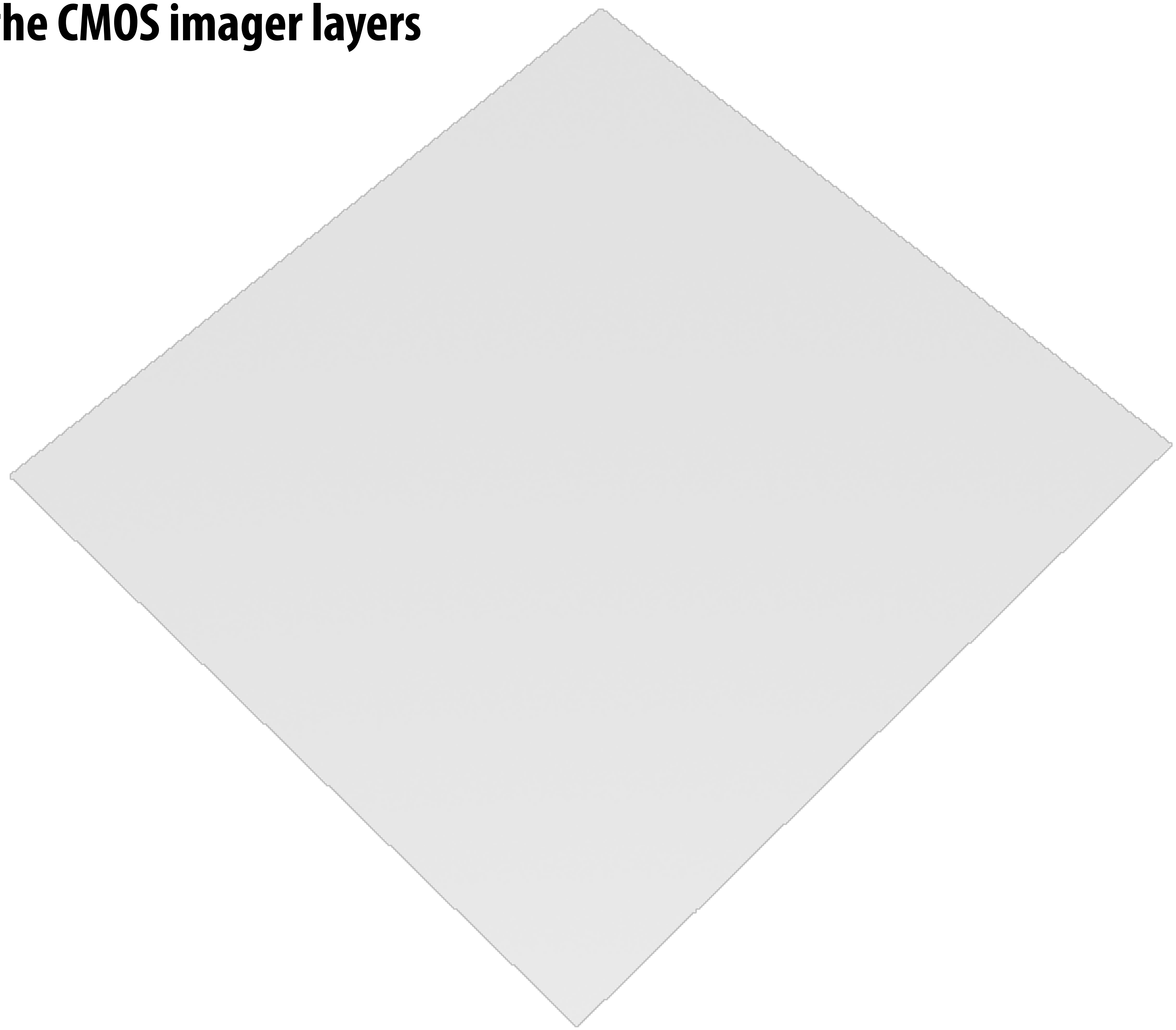
RAW output of sensor

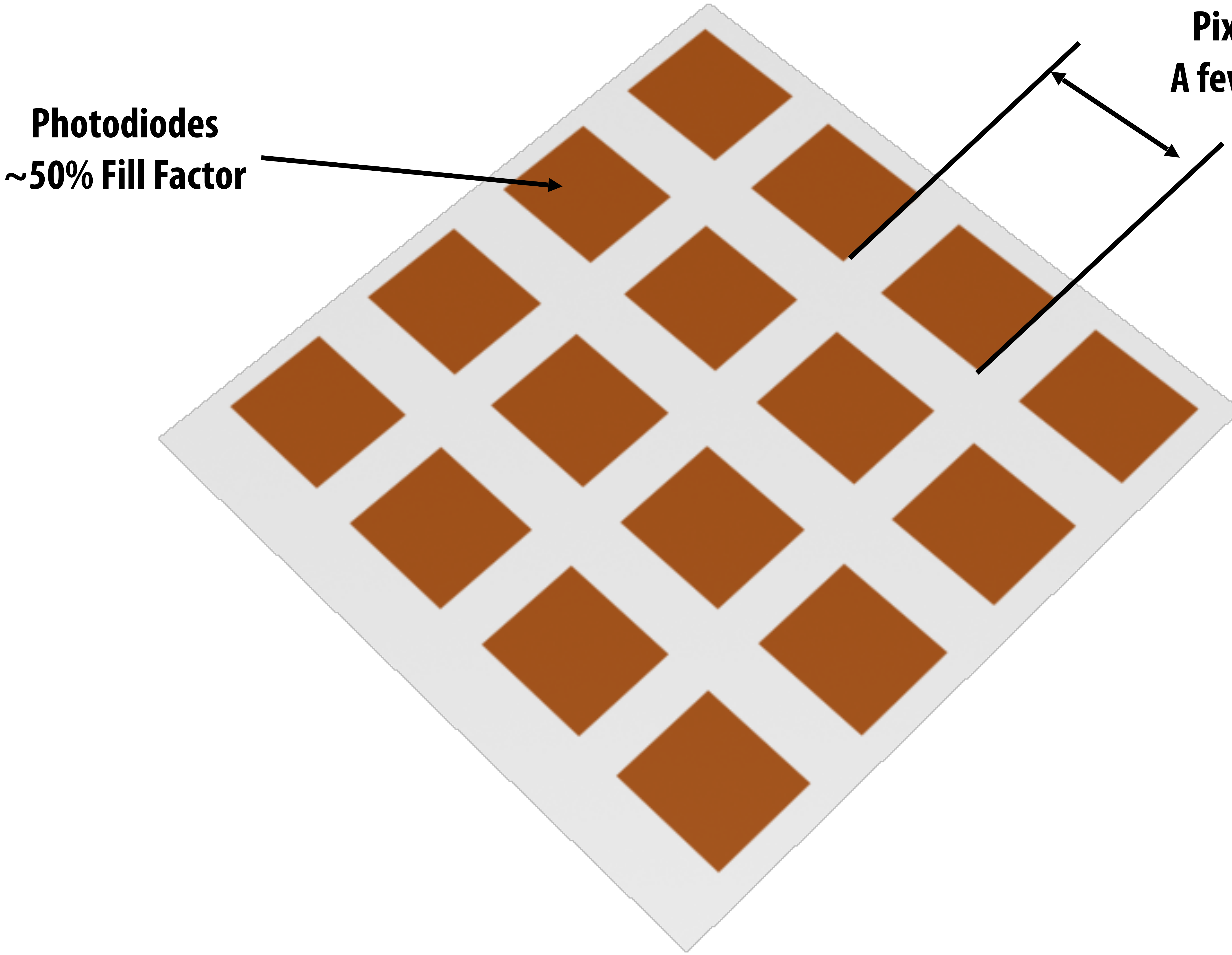


CMOS Pixel Structure

Front-side-illuminated (FSI) CMOS

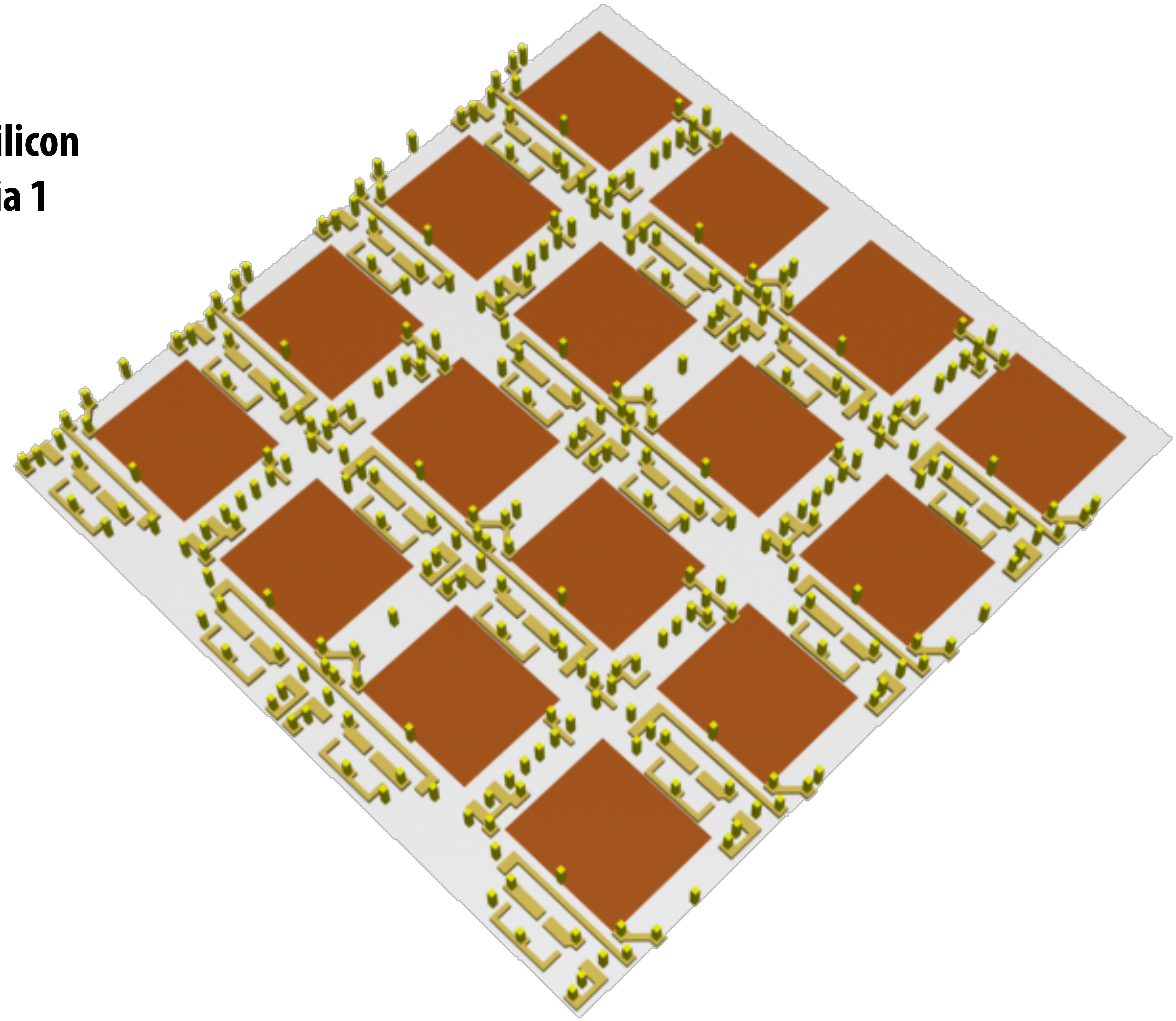
Building up the CMOS imager layers





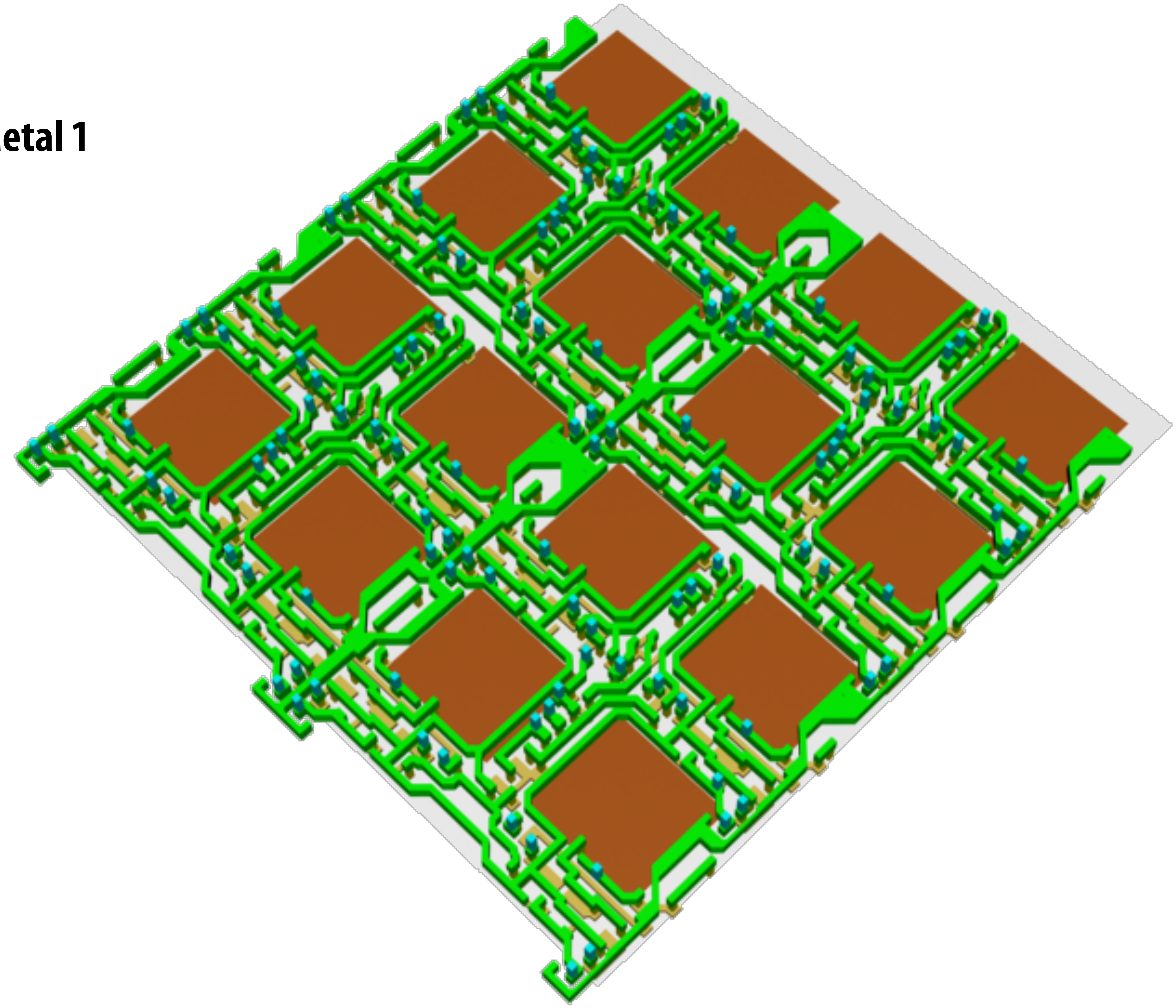
Courtesy R. Motta, Pixim

Polysilicon & Via 1



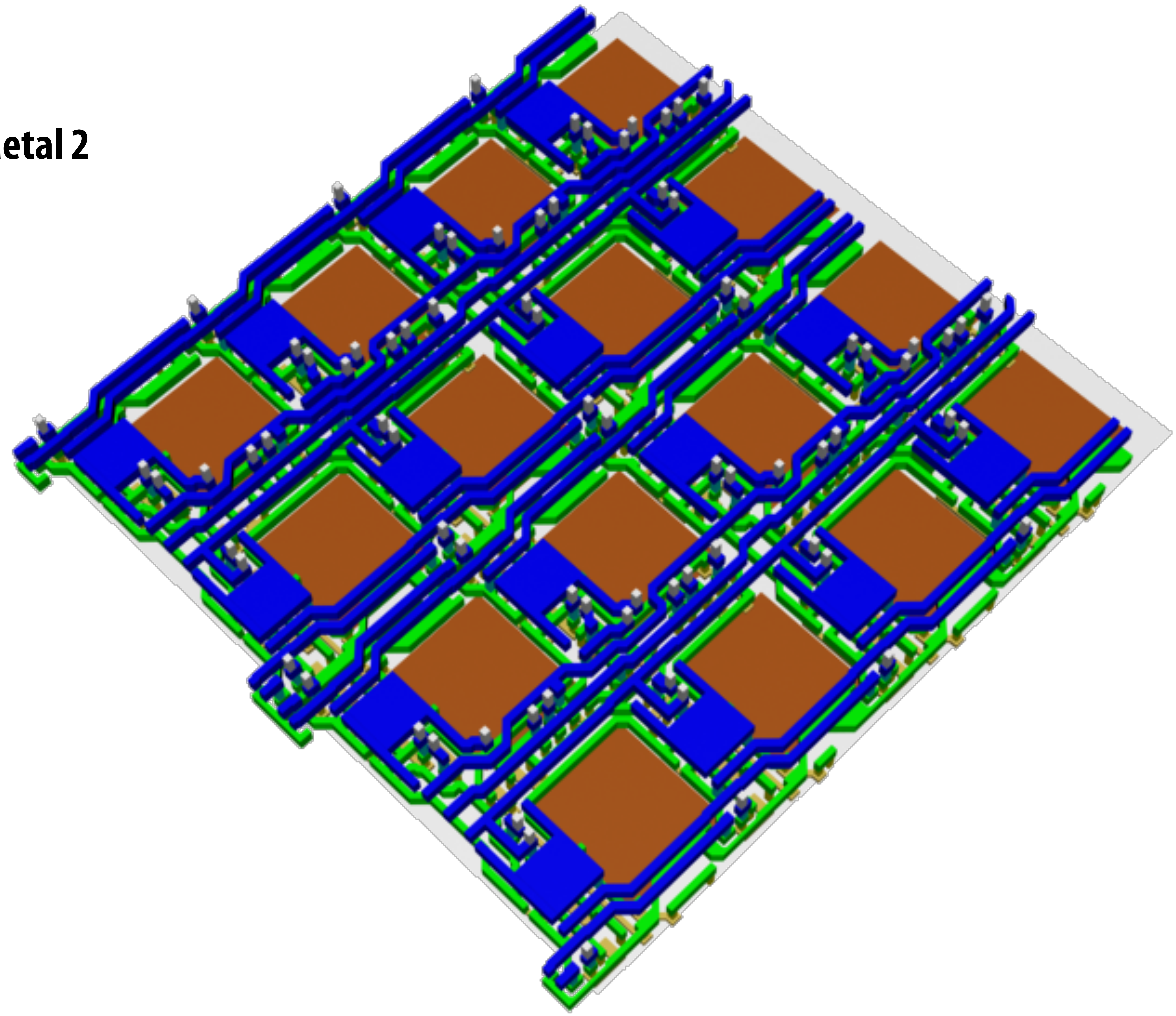
Courtesy R. Motta, Pixim

Metal 1



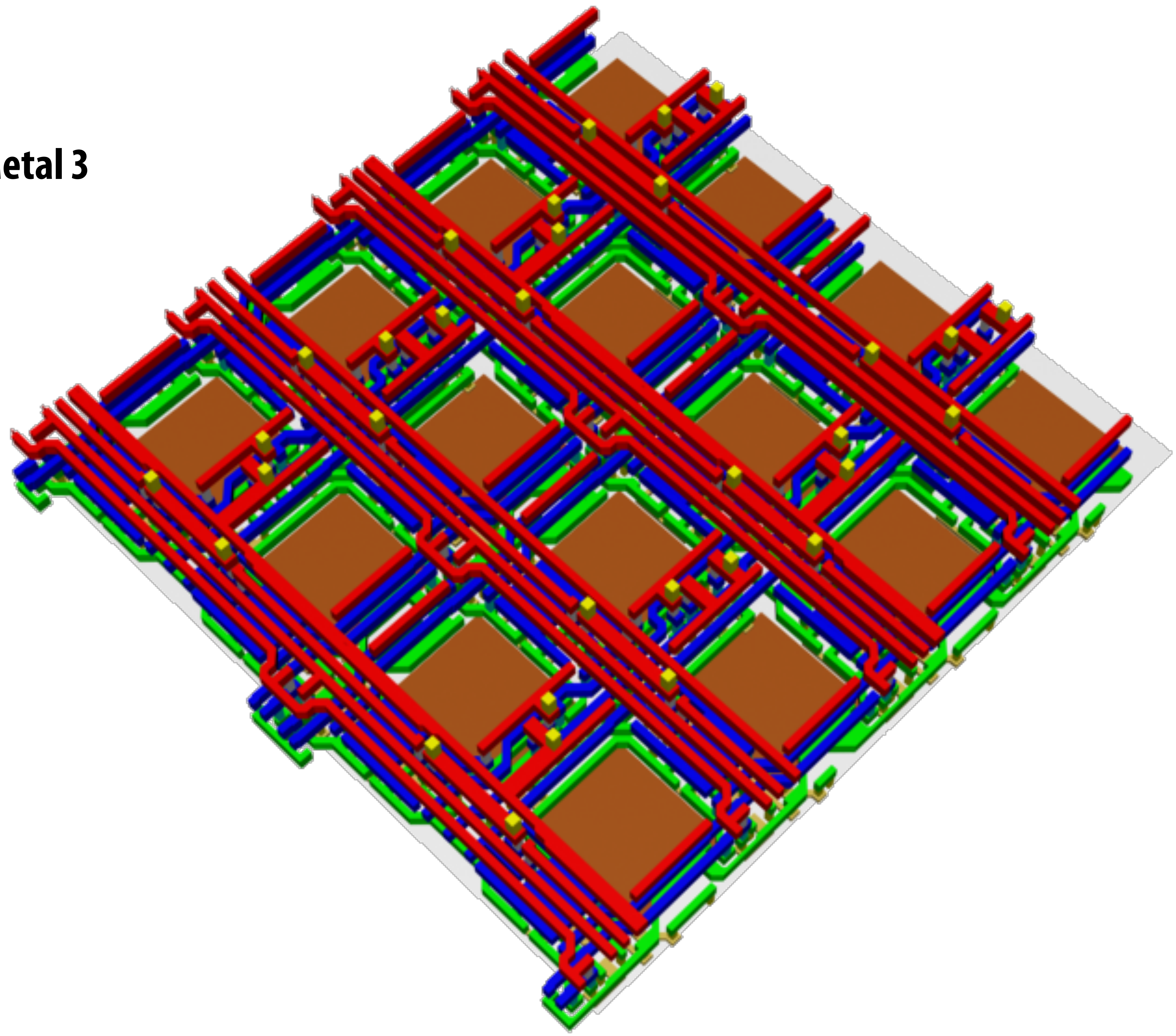
Courtesy R. Motta, Pixim

Metal 2

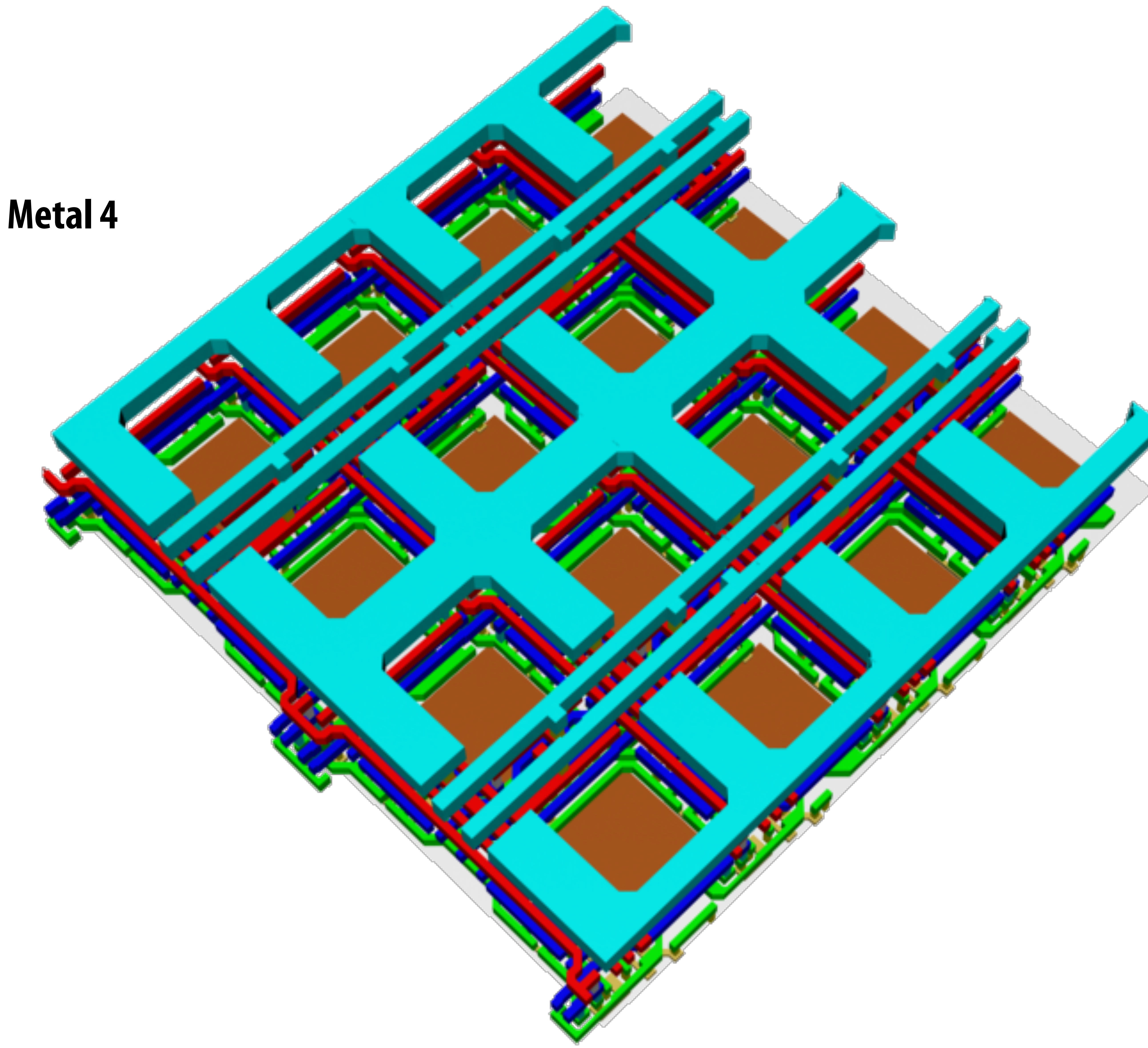


Courtesy R. Motta, Pixim

Metal 3

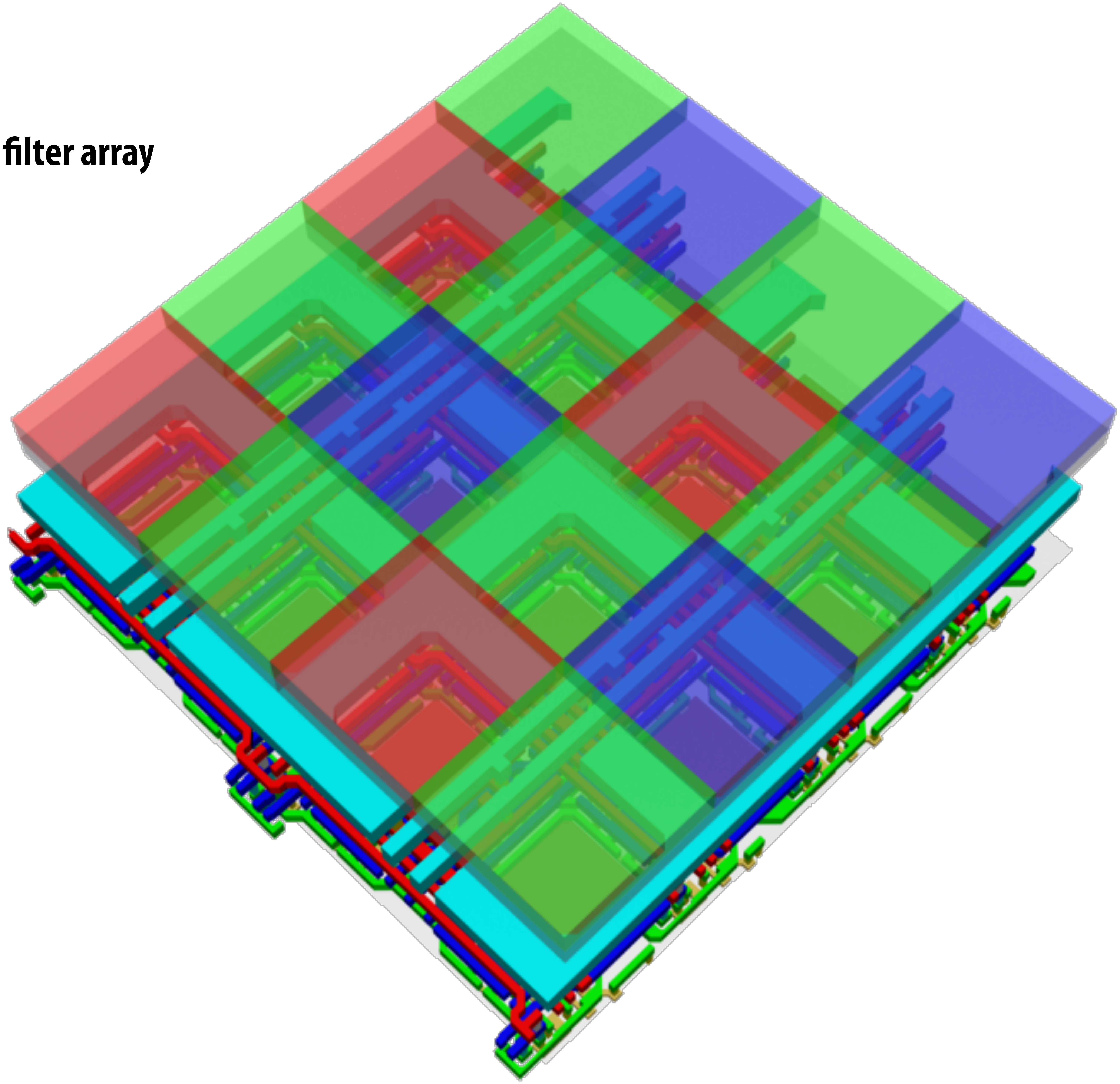


Courtesy R. Motta, Pixim



Courtesy R. Motta, Pixim

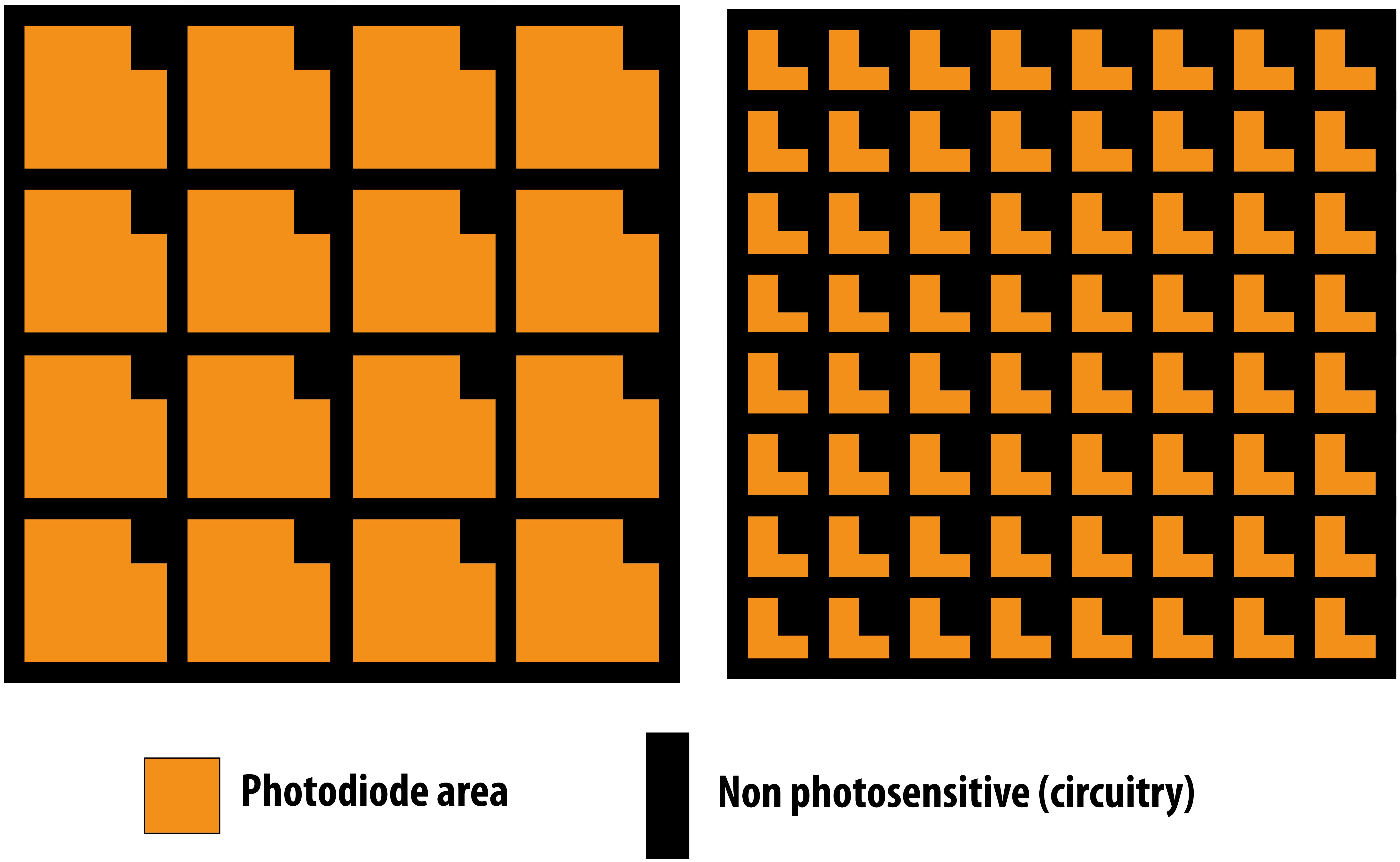
Color filter array



Courtesy R. Motta, Pixim

Pixel fill factor

Fraction of pixel area that integrates incoming light



Slide credit: Ren Ng

CMOS sensor pixel

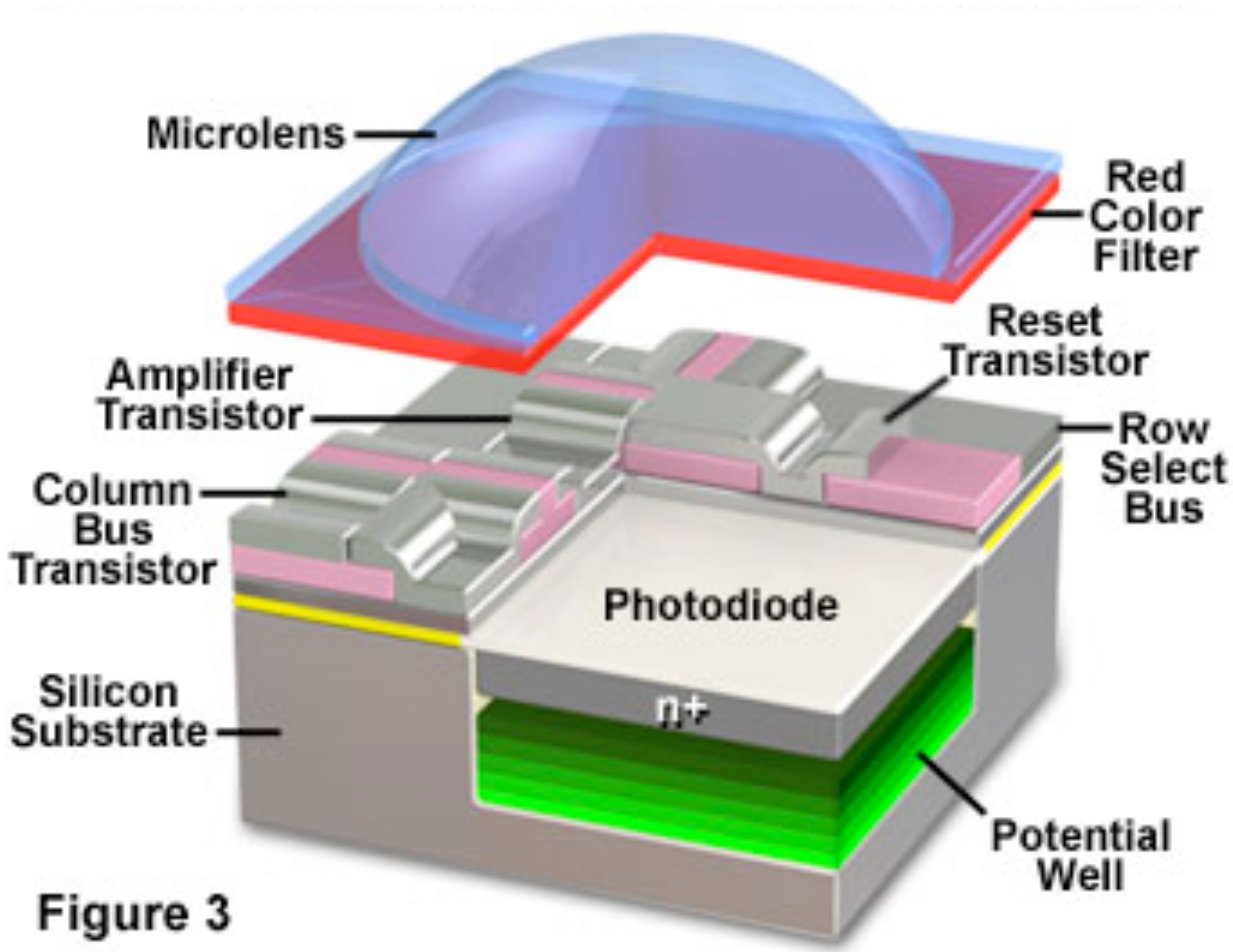


Figure 3

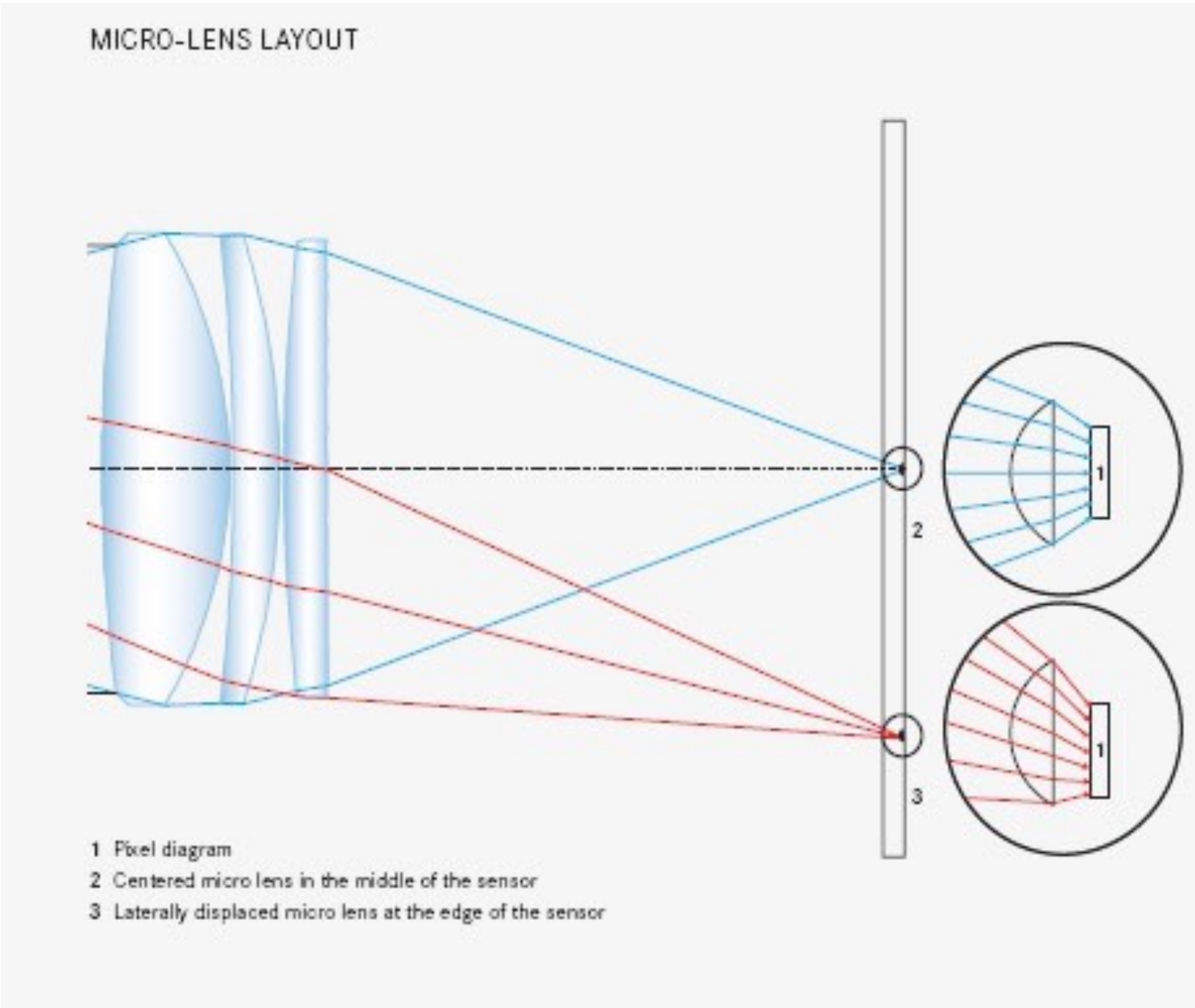
Color filter attenuates light

Microlens (a.k.a. lenslet) steers light toward photo-sensitive region (increases light-gathering capability)

Advanced question: Microlens also serves to reduce aliasing signal. Why?

Illustration credit: Molecular Expressions (http://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html)

Using micro lenses to improve fill factor

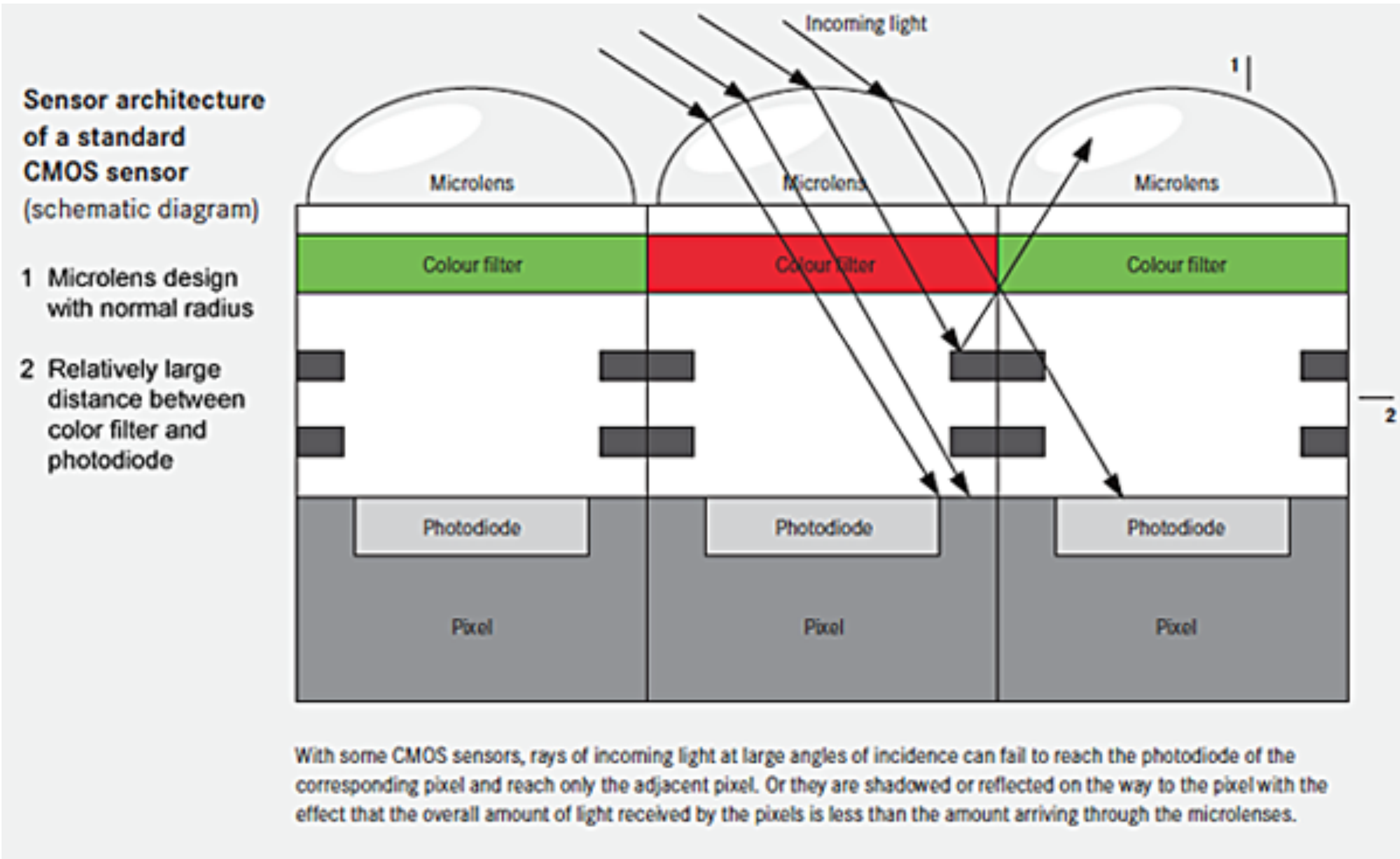


Shifted microlenses on M9 sensor.

Leica M9

Slide credit: Ren Ng

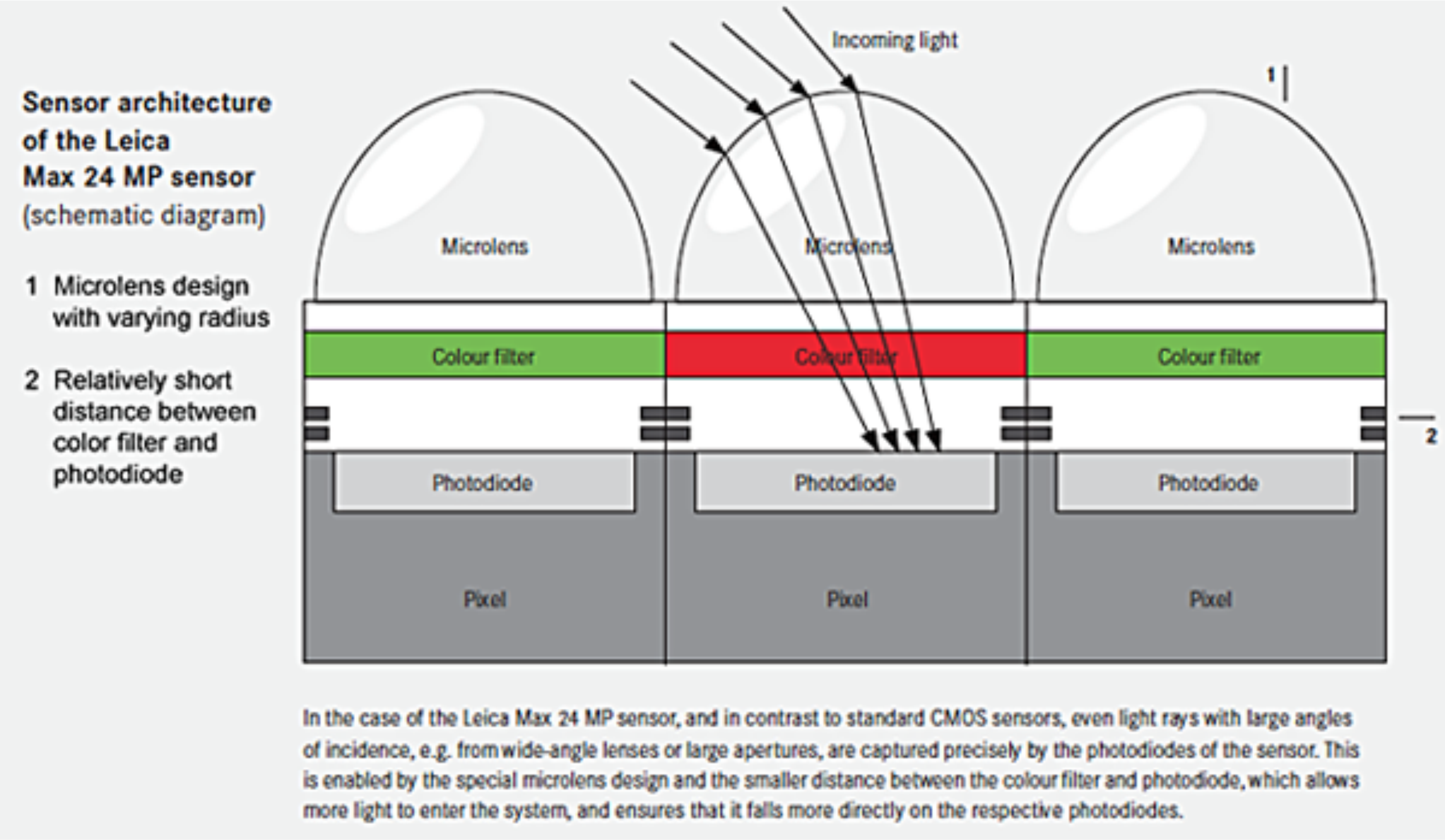
Optical cross-talk



Slide credit: Ren Ng

http://gmpphoto.blogspot.com/2012/09/the-new-leica-max-24mp-cmos-sensor.html

Pixel optics for minimizing cross-talk



Slide credit: Ren Ng

http://gmpphoto.blogspot.com/2012/09/the-new-leica-max-24mp-cmos-sensor.html

Backside illumination sensor

- Traditional CMOS: electronics block light
- Idea: move electronics underneath light gathering region
  - Increases fill factor
  - Reduces cross-talk due since photodiode closer to microns
  - Implication 1: better light sensitivity at fixed sensor size
  - Implication 2: equal light sensitivity at smaller sensor size (shrink sensor)

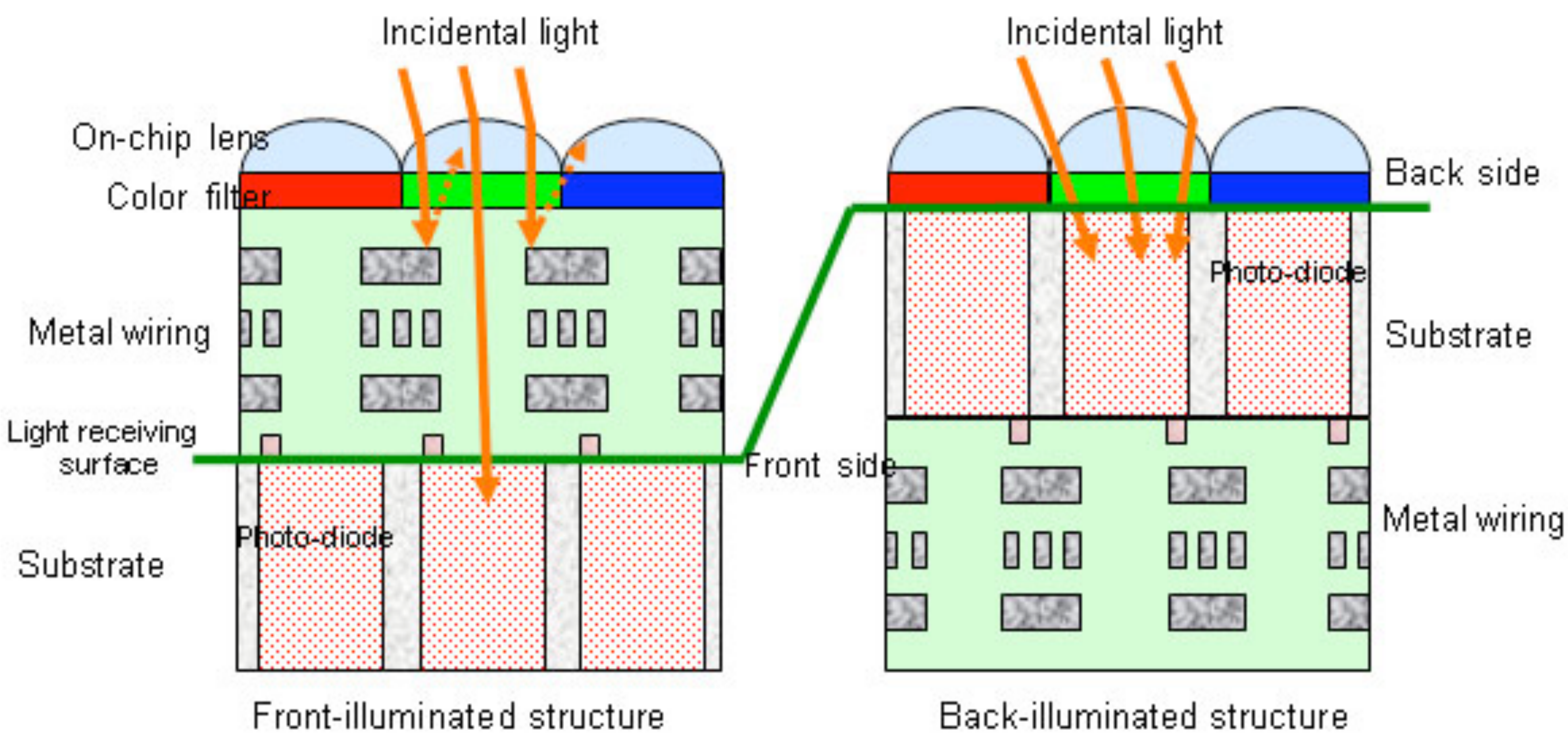


Illustration credit: Sony

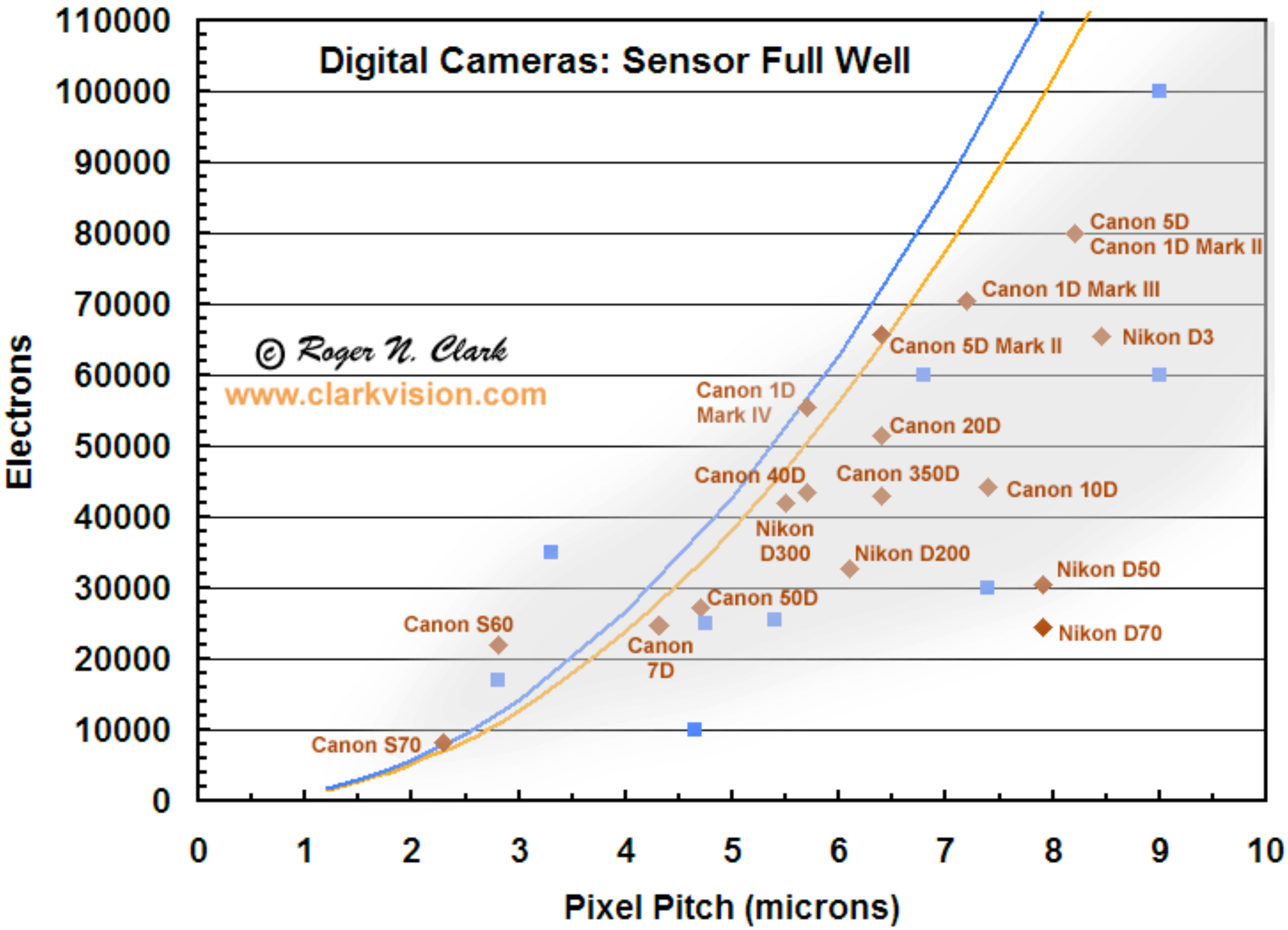
Pixel saturation and noise

Saturated pixels

Photon count for pixels has saturated (no detail in image)

Full-well capacity

Pixel saturates when photon capacity is exceeded

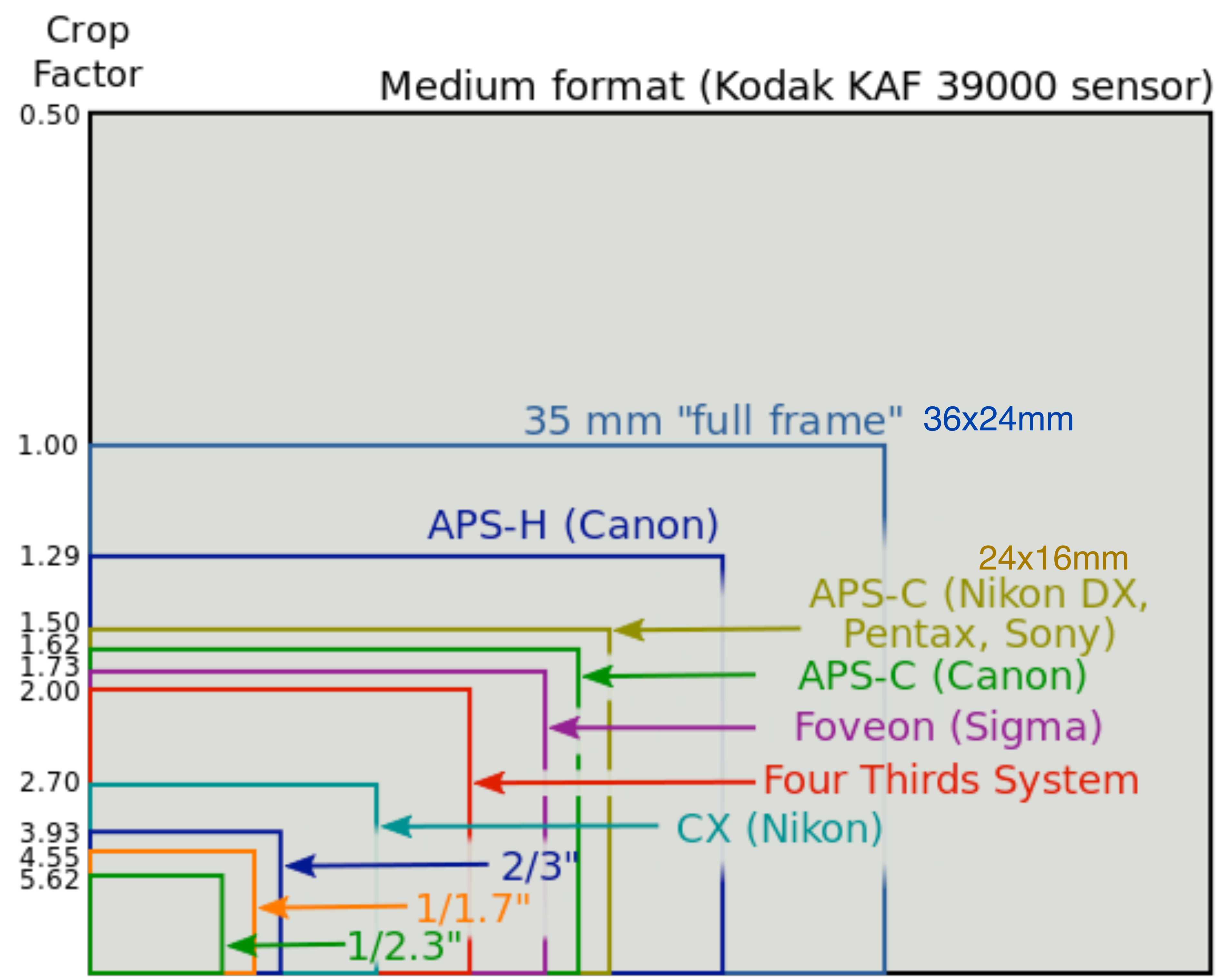


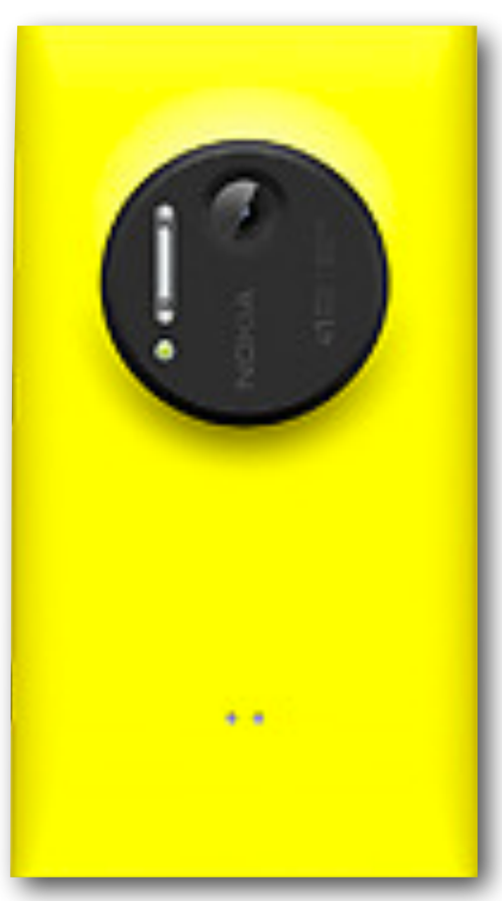
Saturated pixels

Graph credit: clarkvision.com

Bigger sensors = bigger pixels (or more pixels?)

- iPhone X (1.2 micron pixels, 12 MP)
- My Nikon D7000 (APS-C)
  (4.8 micron pixels, 16 MP)
- Nikon D4 (full frame sensor)
  (7.3 micron pixels, 16 MP)
- Implication: very high pixel count sensors can be built with current CMOS technology
  - Full frame sensor with iPhone X pixel size ~ 600 MP sensor

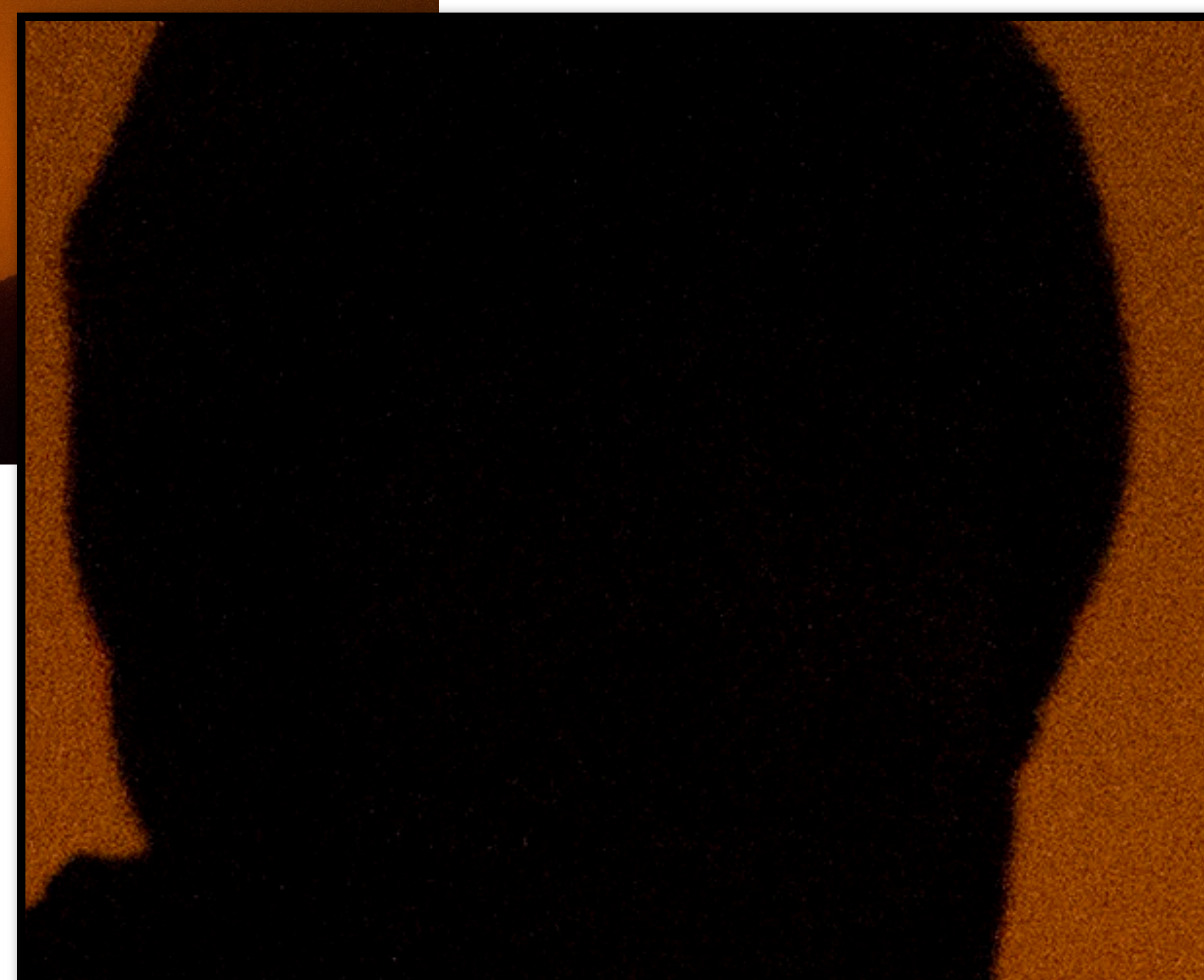




Nokia Lumia
(41 MP)

Image credit: Wikipedia

Measurement noise



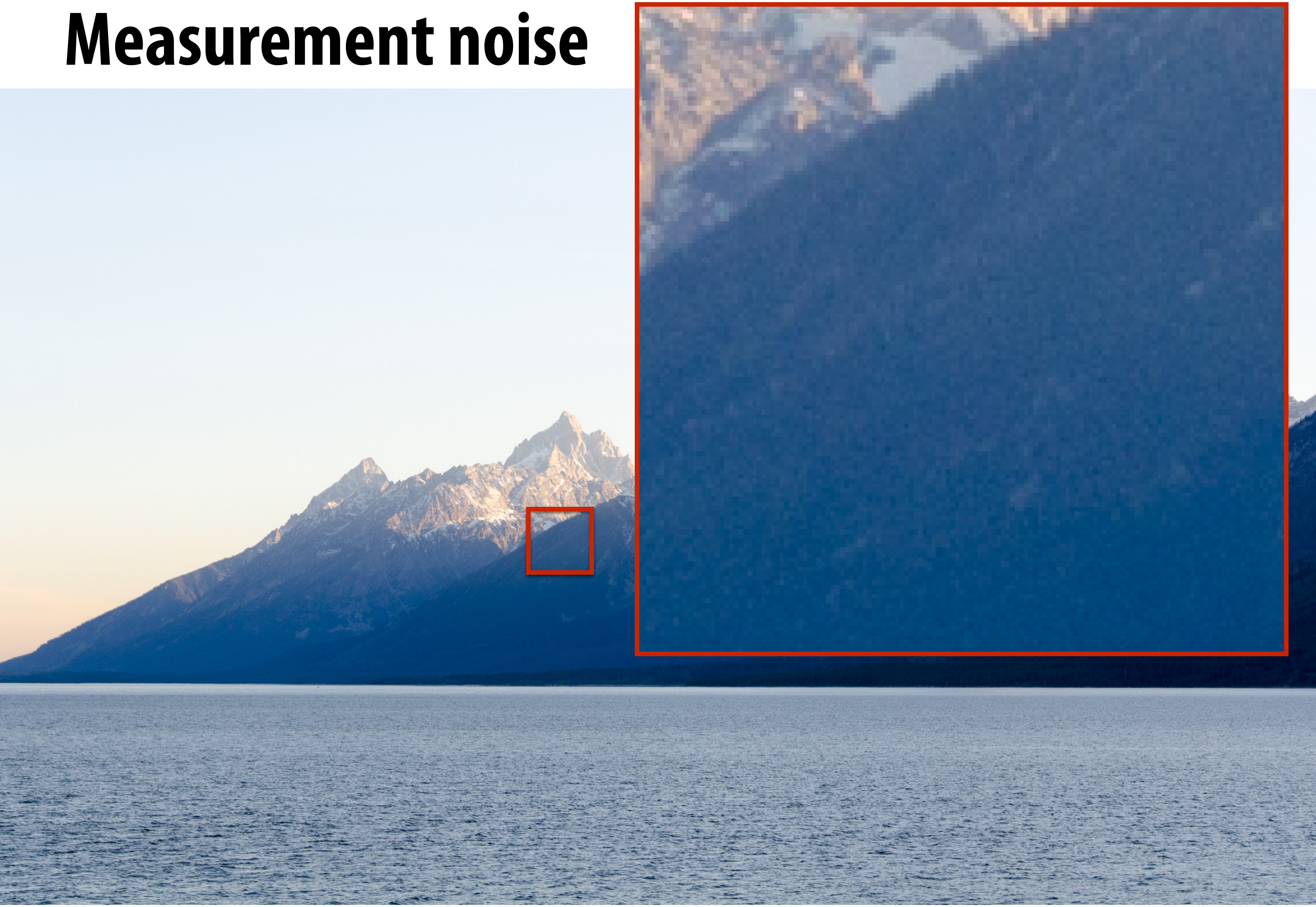
We've all been frustrated by noise in low-light photographs

(or in shadows in daytime images)

Measurement noise

Grand Teton National Park

Measurement noise



Grand Teton National Park

Sources of measurement noise

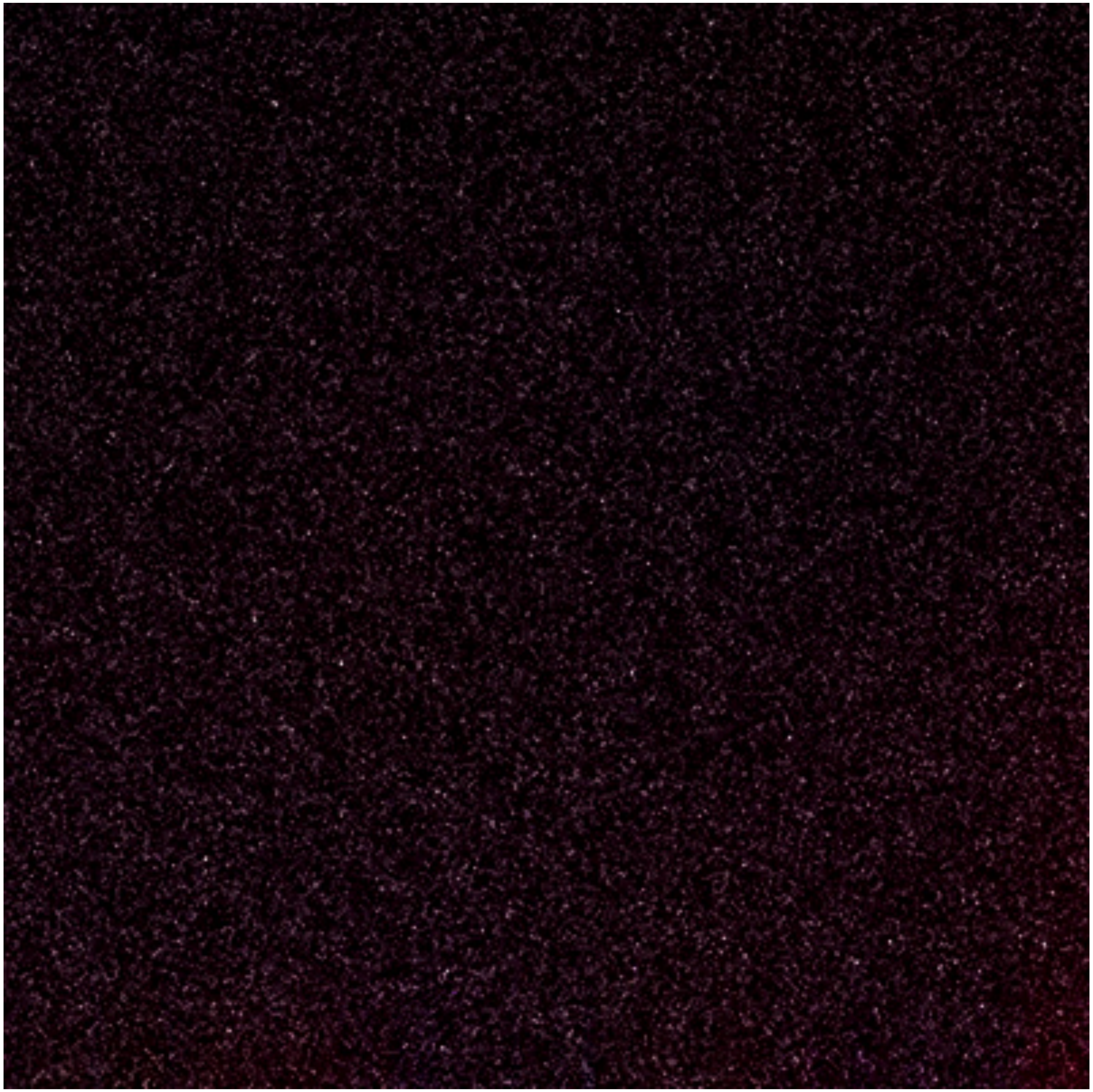
- Photon shot noise:
  - Photon arrival rate takes on Poisson distribution
  - Standard deviation = sqrt(N)   (N = number of photon arrivals)
  - Signal-to-noise ratio (SNR): N/sqrt(N)
  - Implication: brighter the signal, the higher the SNR
- Dark-shot noise
  - Due to leakage current in sensor
  - Electrons dislodged due to thermal activity (increases exponentially with sensor temperature)
- Non-uniformity of pixel sensitivity (due to manufacturing defects)
- Read noise
  - e.g., due to amplification / ADC

Addressed by: subtract dark image

Addressed by: subtract flat field image (e.g., image of gray wall),

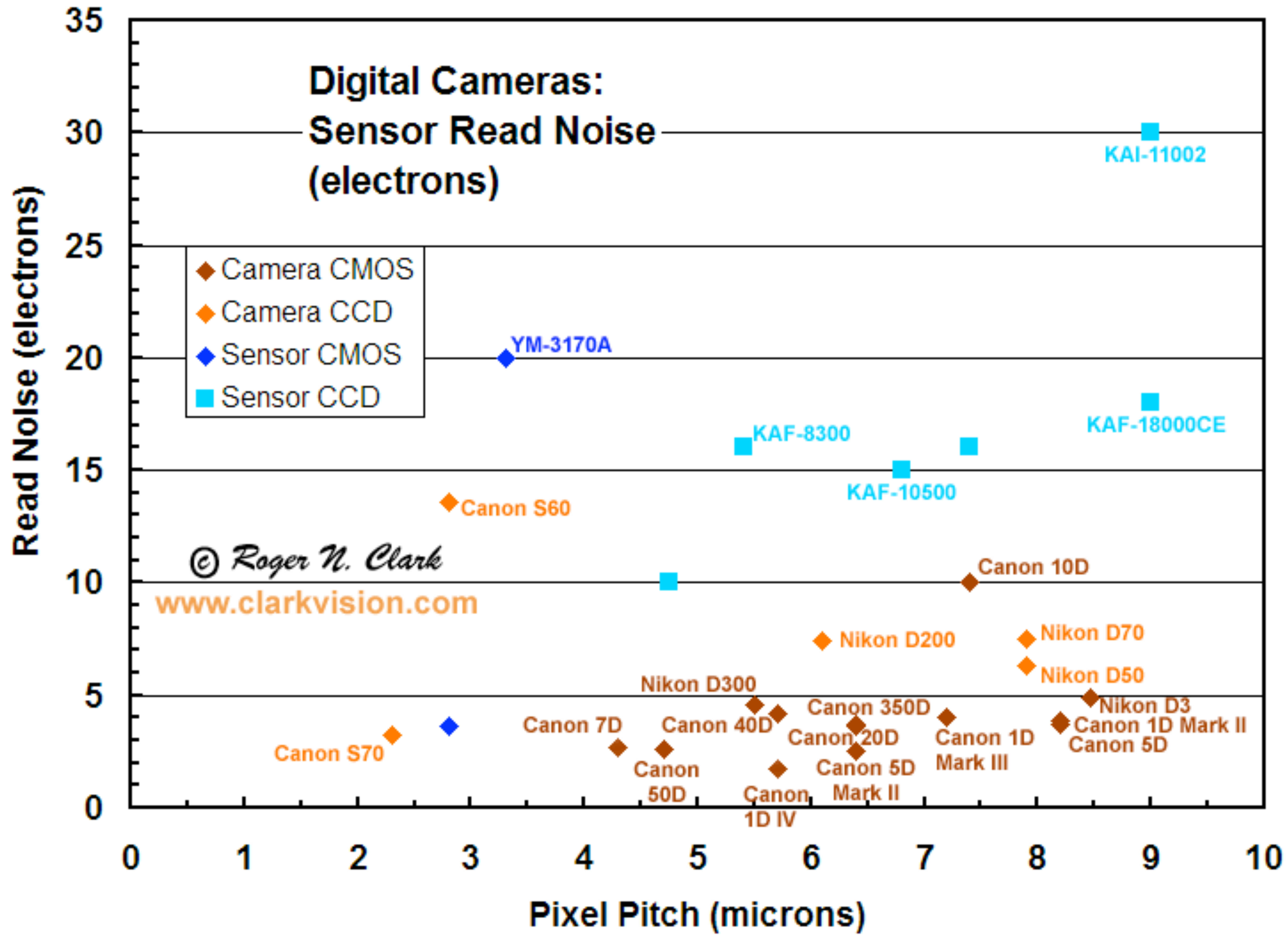
Dark shot noise / read noise

Black image examples: Nikon D7000, High ISO



1 sec exposure

Read noise



Read noise is largely independent of pixel size

Large pixels + bright scene = large N

So, noise determined largely by photon shot noise

Image credit: clarkvision.com

Maximize light gathering capability

- **Goal: increase signal-to-noise ratio**
  - Dynamic range of a pixel (ratio of brightest light measurable to dimmest light measurable) is determined by the noise floor (minimum signal) and the pixel's full-well capacity (maximum signal)
- **Big pixels**
  - Nikon D4: 7.3 um
  - iPhone X: 1.2 um
- **Sensitive pixels**
  - Good materials
  - High fill factor

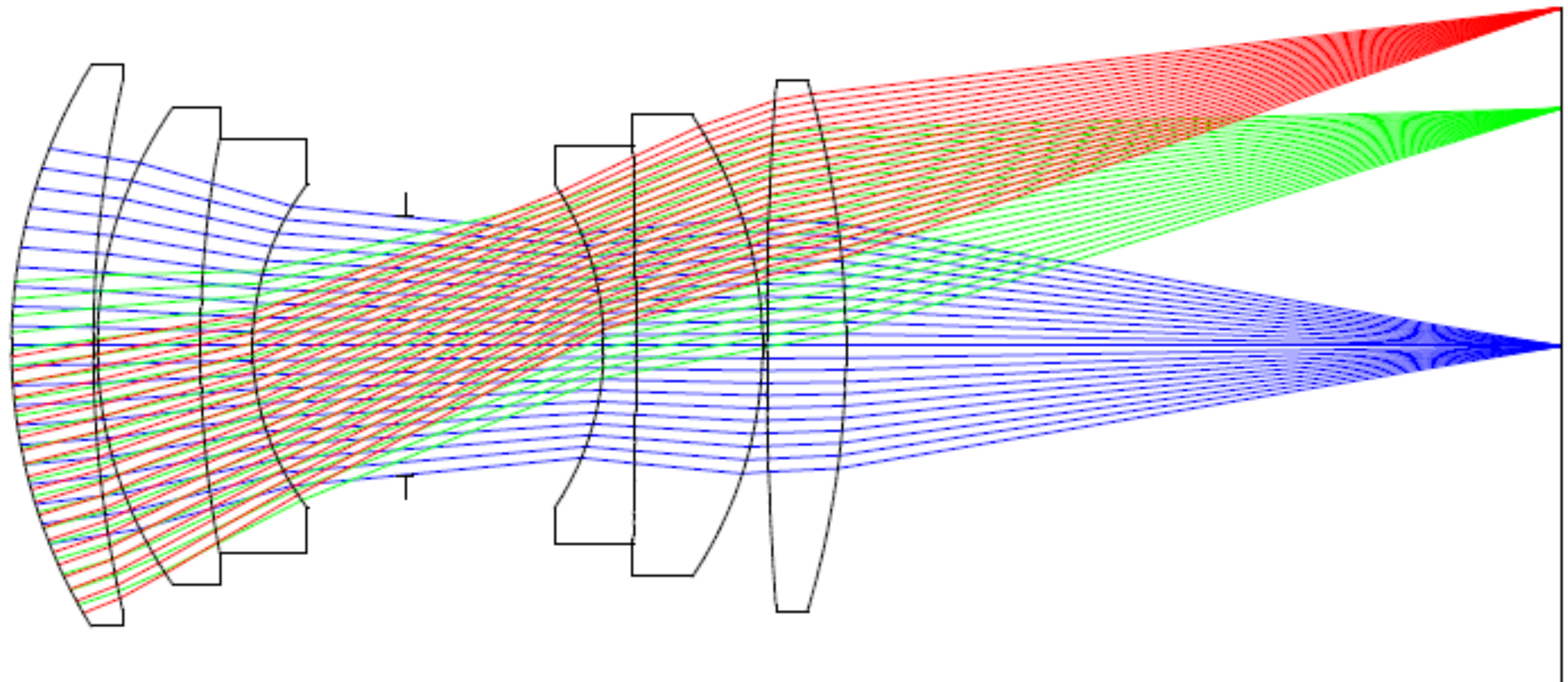
Artifacts arising from lenses

Vignetting

Image of white wall (Note: I contrast-enhanced the image to show effect)

Types of vignetting

Optical vignetting: less light reaches edges of sensor due to physical obstruction in lens



Pixel vignetting: light reaching pixel at an oblique angle is less likely to hit photosensitive region than light incident from straight above (e.g., obscured by electronics)

- **Microlens reduces pixel vignetting**

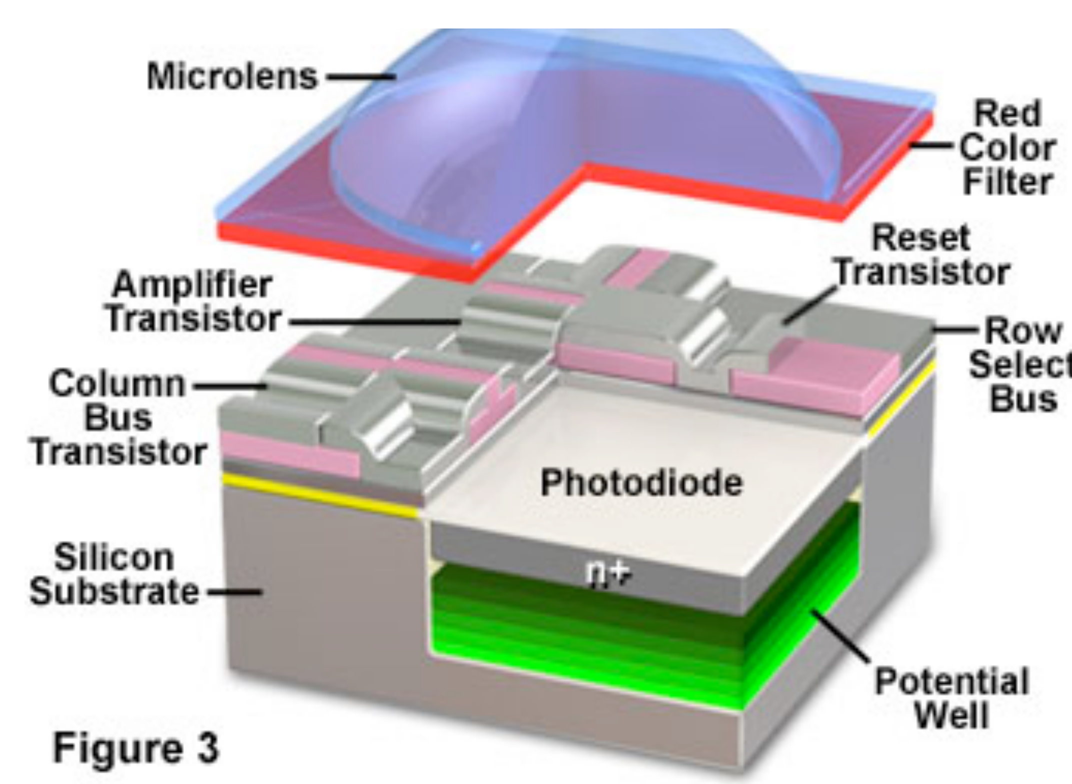
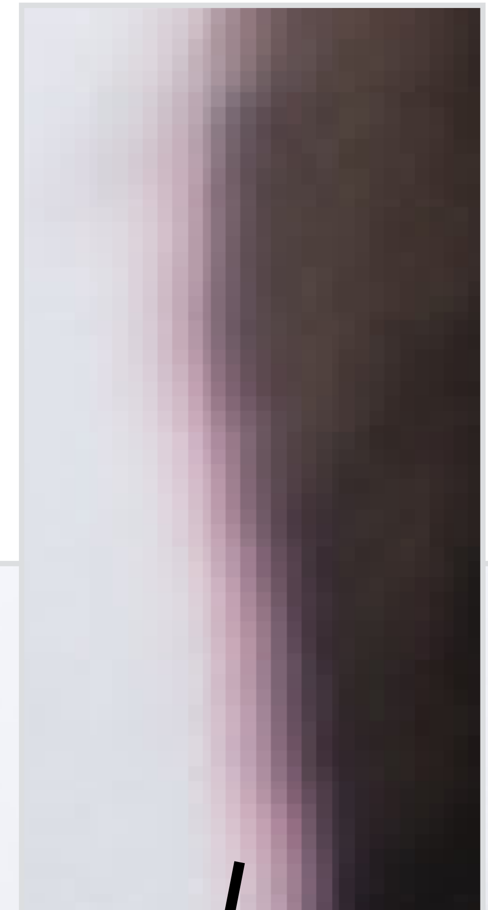


Figure 3

Image credit: Mark Butterworth

Chromatic aberration (due to lens)



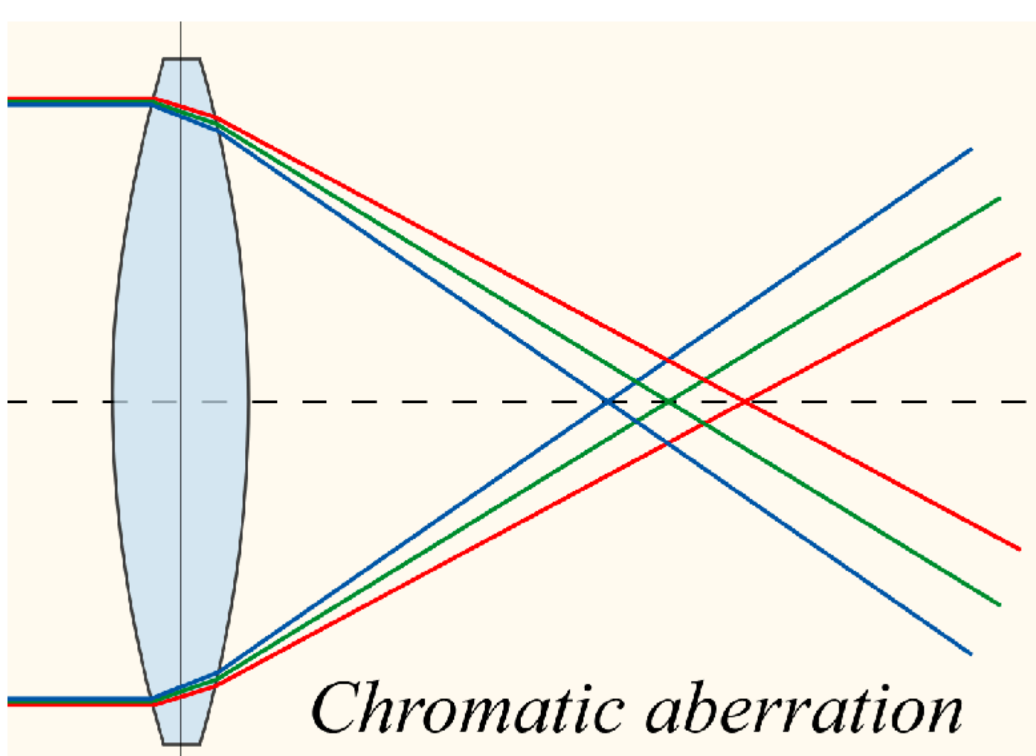
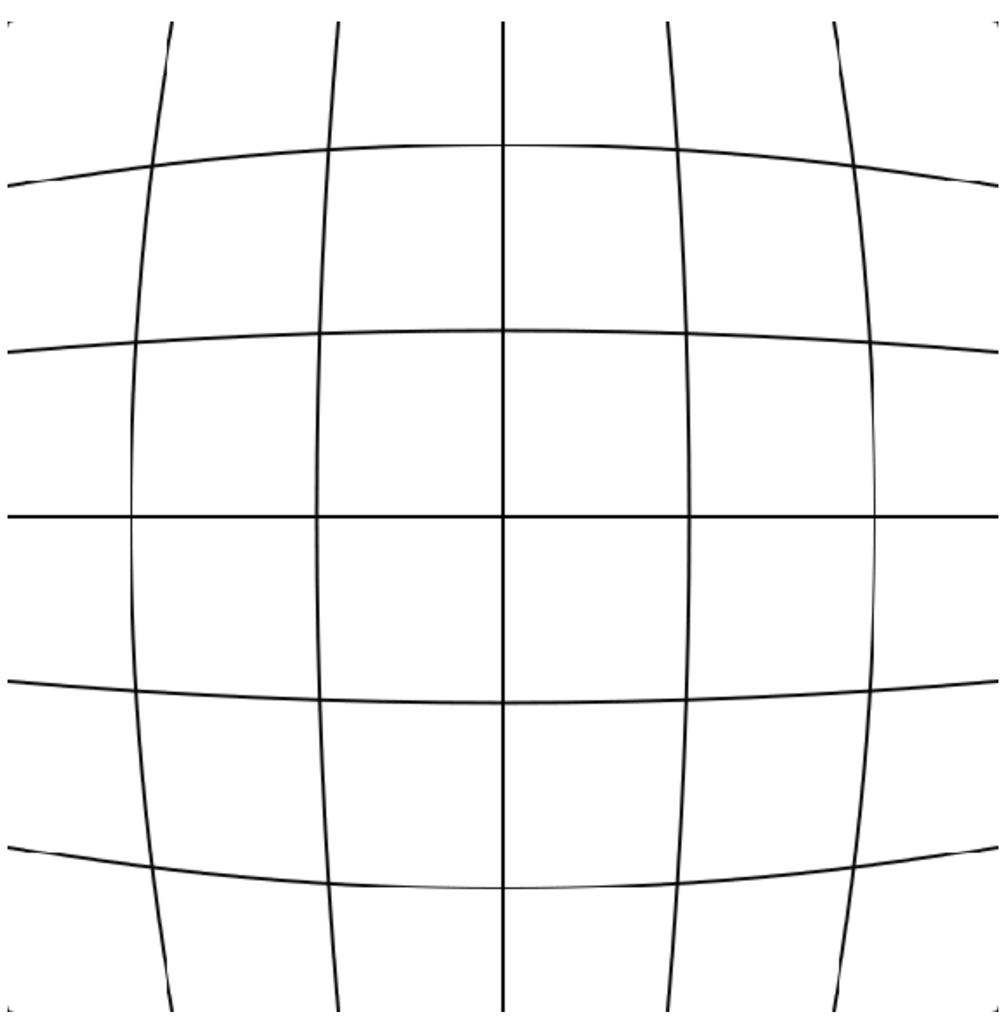


Image credit: Wikipedia

More challenges

- **Chromatic shifts over sensor**
  - Pixel light sensitivity changes over sensor due to interaction with microlens (Index of refraction depends on wavelength, so some wavelengths are more likely to suffer from cross-talk or reflection. Ug!)
- **Lens distortion**



Pincushion distortion

Captured Image

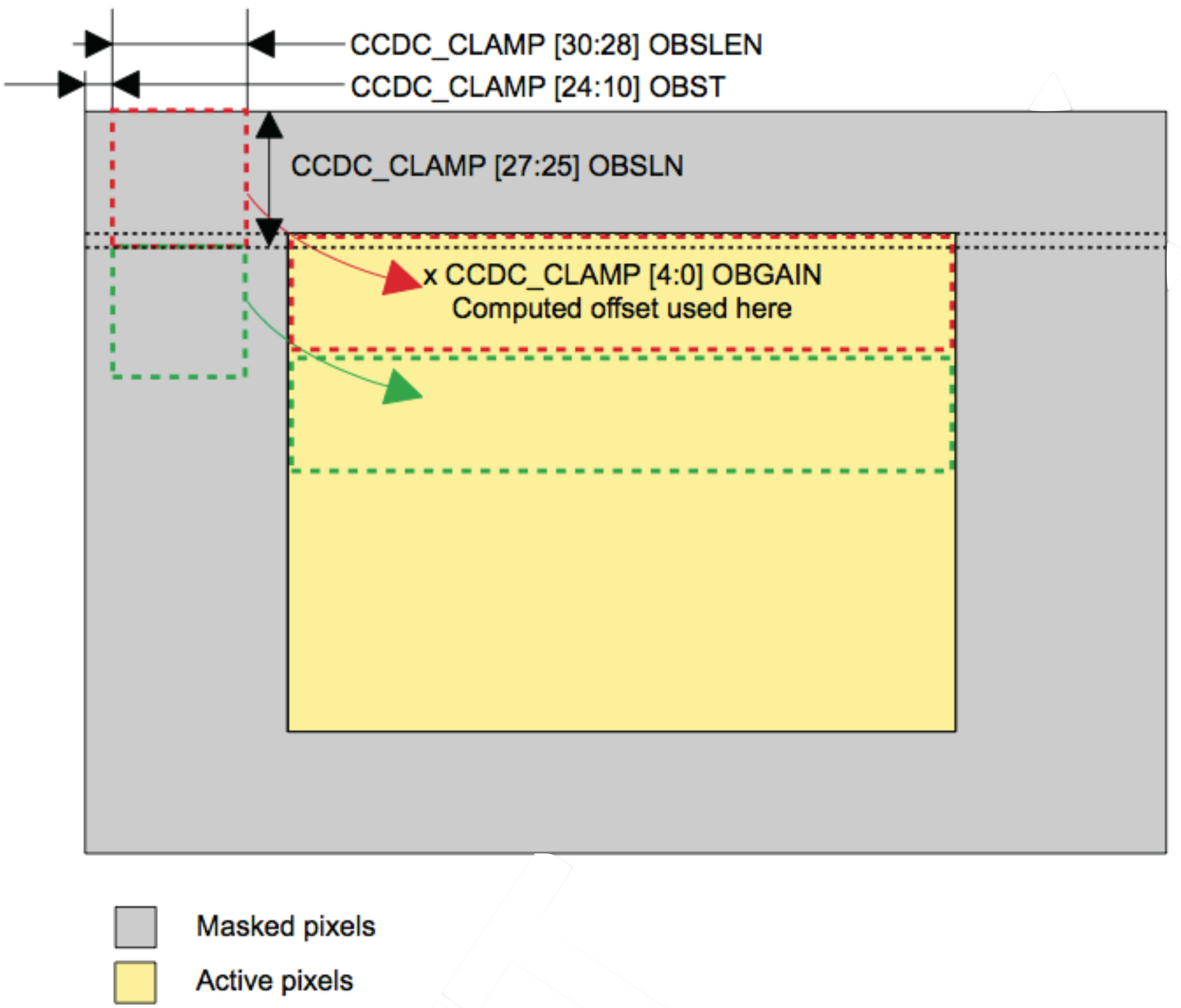
Corrected Image

Image credit: PCWorld

Part 2:
A simple RAW image processing pipeline
(how software takes sensor output to a high-quality RGB image)

Optical clamp: remove sensor offset bias

`output_pixel = input_pixel – [average of pixels from optically black region]`



Remove bias due to sensor black level (from nearby sensor pixels at time of shot)

Correct for defective pixels

- **Store LUT with known defective pixels**
  - e.g., determined on manufacturing line, during sensor calibration and test
- **Example correction methods**
  - Replace defective pixel with neighbor
  - Replace defective pixel with average of neighbors
  - Correct defect by subtracting known bias for the defect

```
output_pixel = (isdefectpixel(current_pixel_xy)) ?
                  average(previous_input_pixel, next_input_pixel) :
                  input_pixel;
```

- **Will describe solutions based only analyzing pixel values (later)**

Lens shading compensation

- **Correct for vignetting**
  - Good implementations will consider wavelength-dependent vignetting (that creates chromatic shift over the image)

- **Possible implementations:**
  - Use flat-field photo stored in memory
    - e.g., lower resolution buffer, upsampled on-the-fly
    - Use analytic function to model correction

```
gain = upsample_compensation_gain_buffer(current_pixel_xy);
output_pixel = gain * input_pixel;
```

White balance

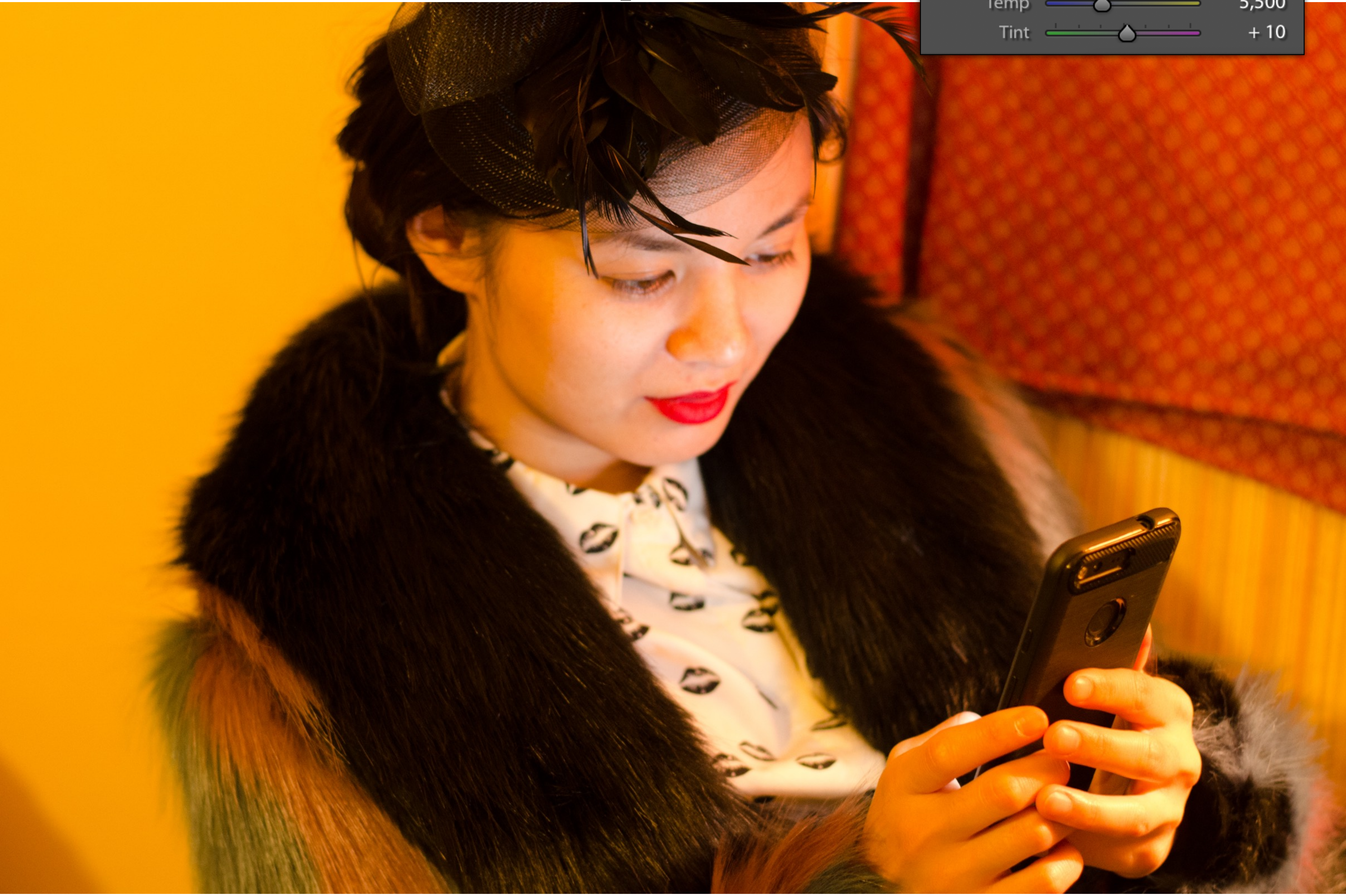
- Adjust relative intensity of rgb values (so neutral tones appear neutral)

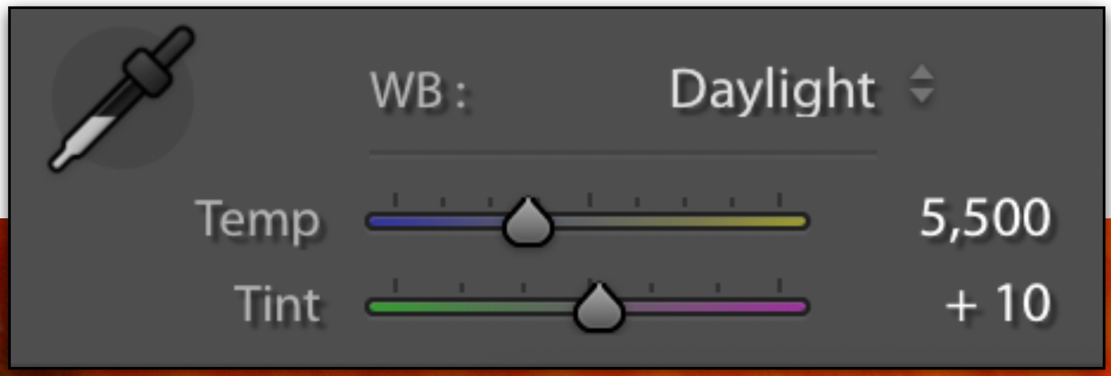
```
output_pixel = white_balance_coeff * input_pixel
// note: in this example, white_balance_coeff is vec3
// (adjusts ratio of red-blue-green channels)
```

- The same "white" object will generate different sensor response when illuminated by different spectra. Camera needs to infer what the lighting in the scene was.

Image credit: basedigitalphotography.com

White balance example





White balance example

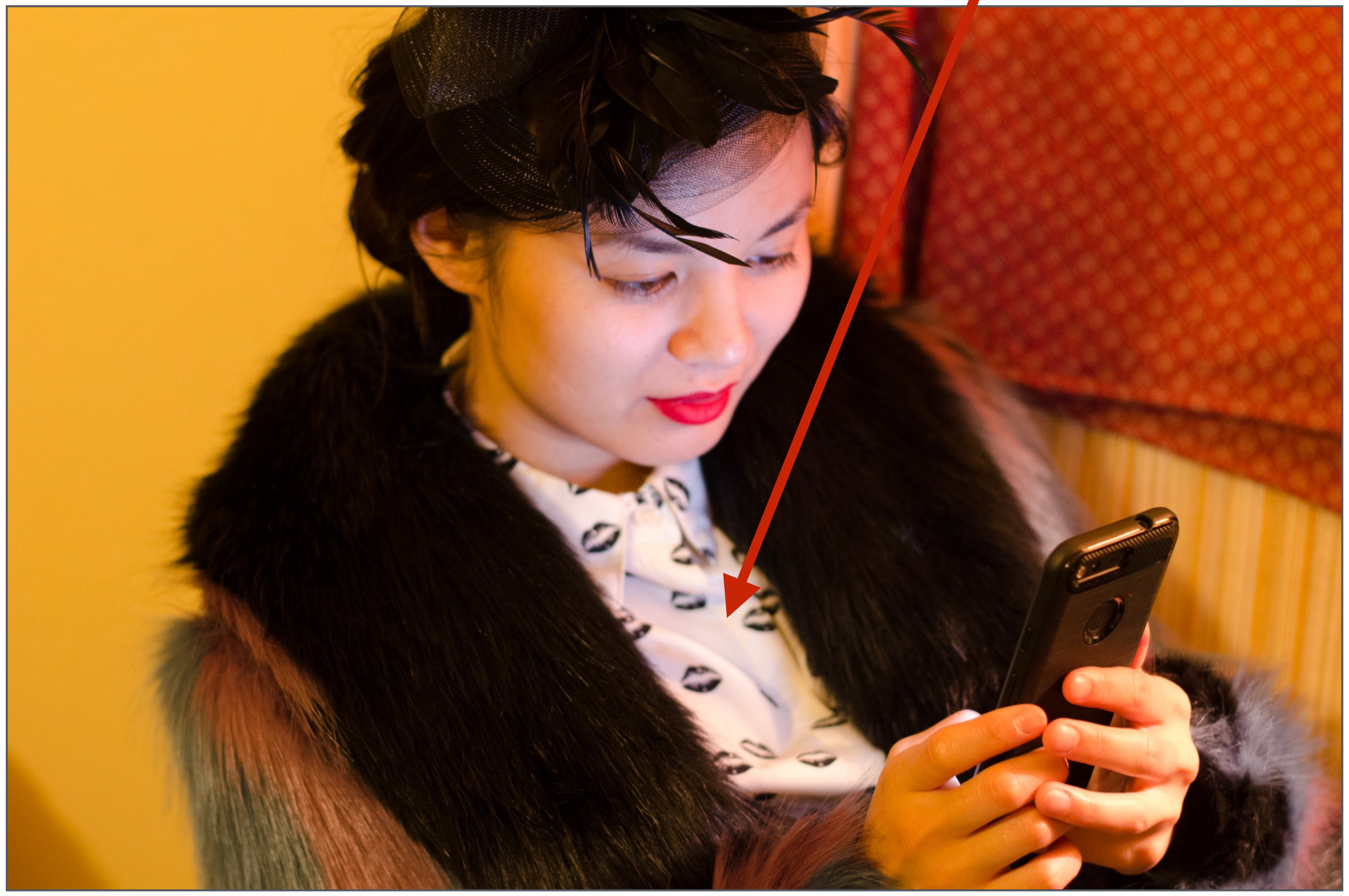
White balance example

White balance algorithms

- White balance coefficients depend on analysis of image contents
  - Calibration based: get value of pixel of "white" object: (r_w, g_w, b_w)
    - Scale all pixels by $(1/r_w, 1/g_w, 1/b_w)$
  - Heuristic based: camera must guesse which pixels correspond to white objects in scene
    - Gray world assumption: make average of all pixels in image gray
    - Brightest pixel assumption: find brightest region of image, make it white ([1,1,1])

- Modern white-balance algorithms are based on learning correct scaling from examples
  - Create database of images for which good white balance settings are known (e.g., manually set by human)
  - Learning mapping from image features to white balance settings
  - When new photo is taken, use learned model to predict good white balance settings

Scale r,g,b values so these pixels are (1,1,1)



Demosiac

- Produce RGB image from mosaiced input image
- Basic algorithm: bilinear interpolation of mosaiced values (need 4 neighbors)
- More advanced algorithms:
  - Bicubic interpolation (wider filter support region… may overblur)
  - Good implementations attempt to find and preserve edges in photo

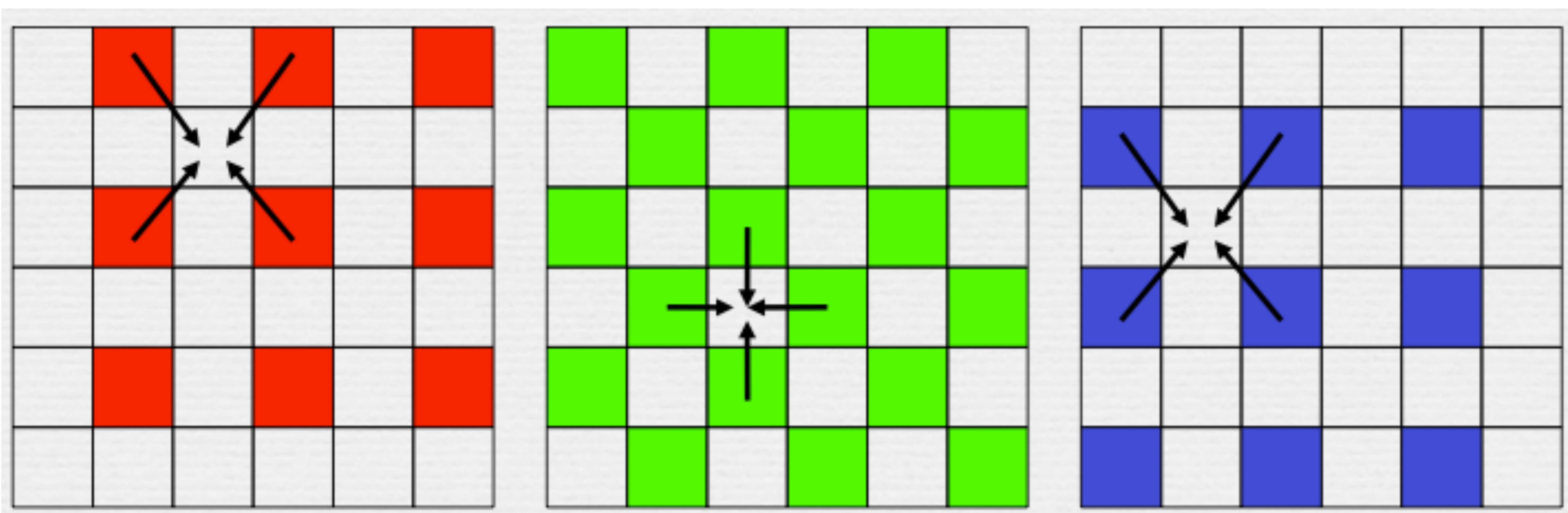
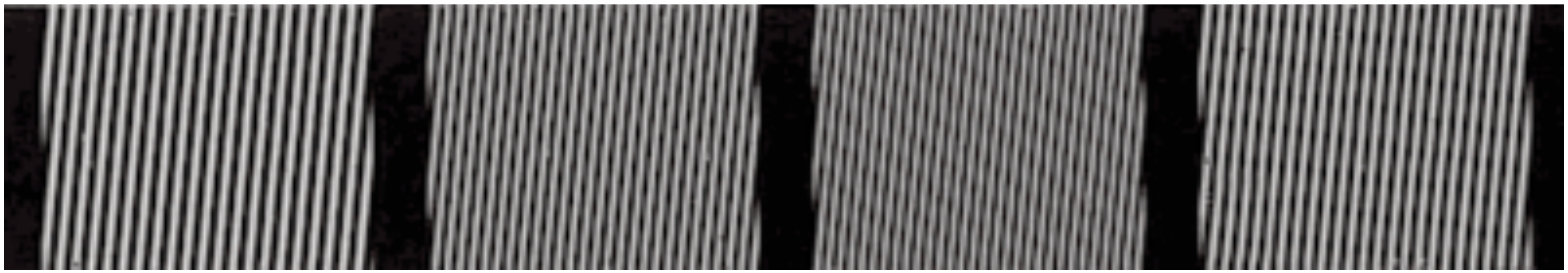


Image credit: Mark Levoy

Demosaicing errors

- Common difficult case: fine diagonal black and white stripes
- Result: moire pattern color artifacts

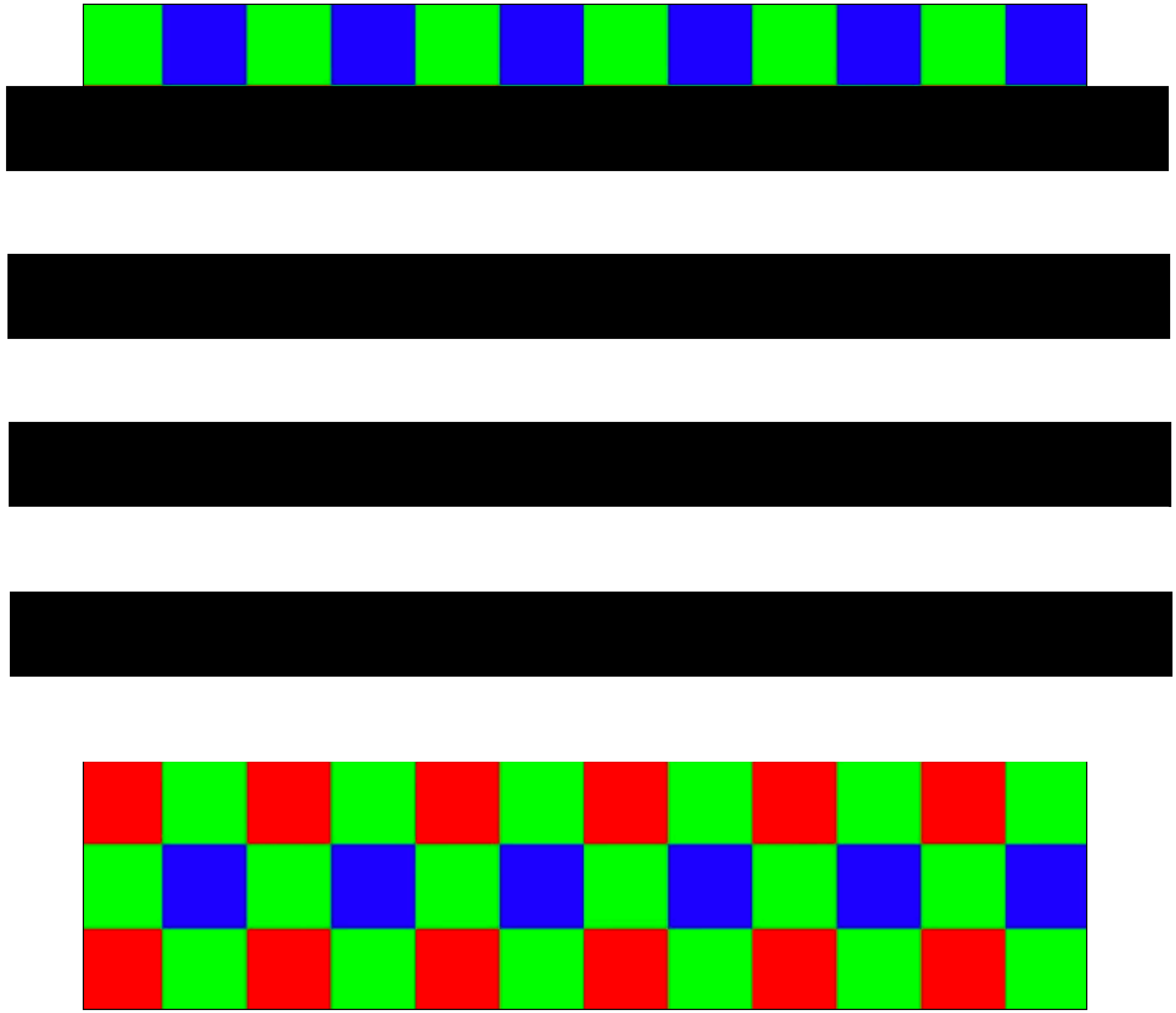


RAW data from sensor

RGB result after demosaic

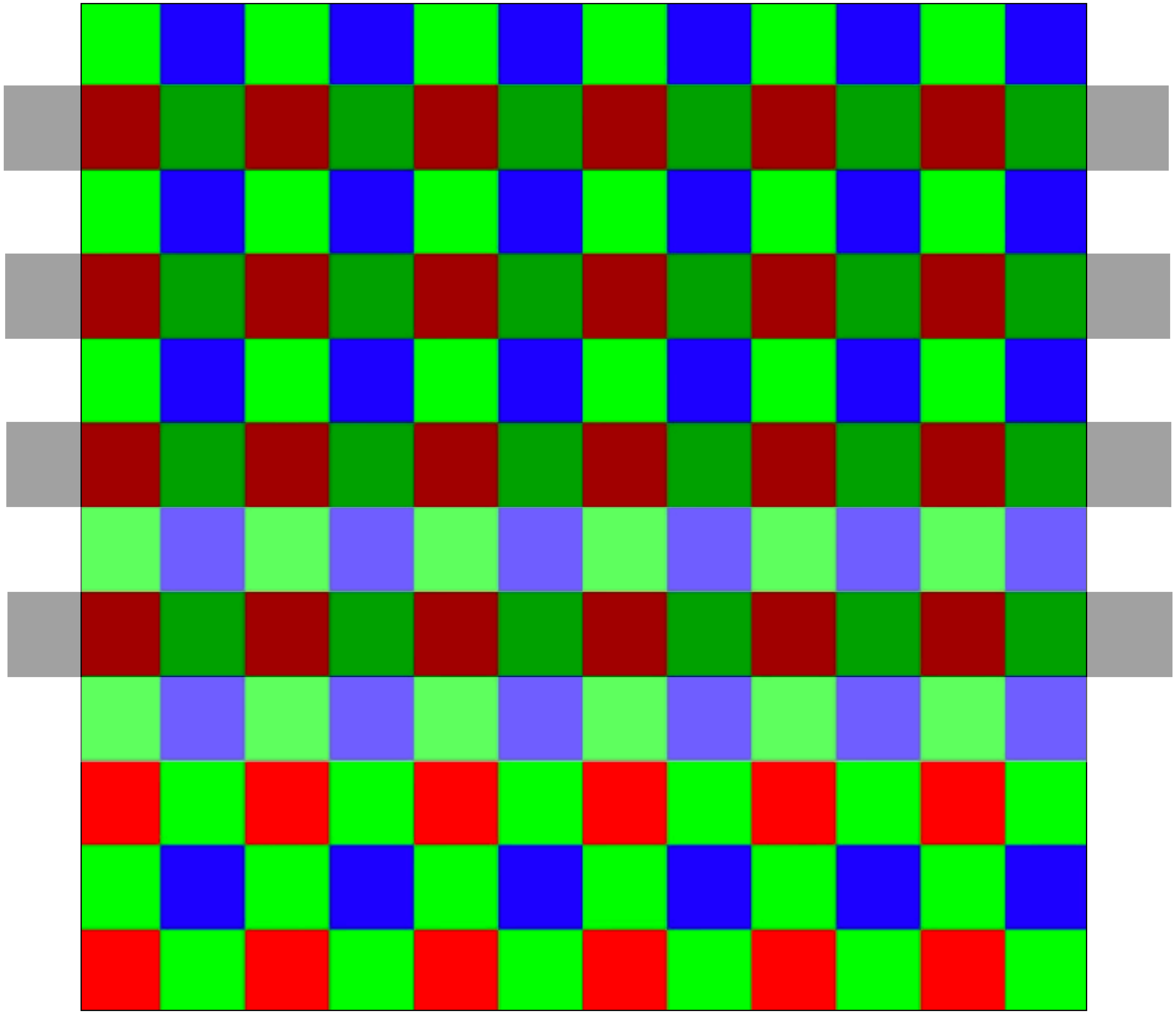
Image credit: http://almanazir.typepad.com/almanazir/2006/11/how_a_camera_ph_1.html

Demosaicing errors



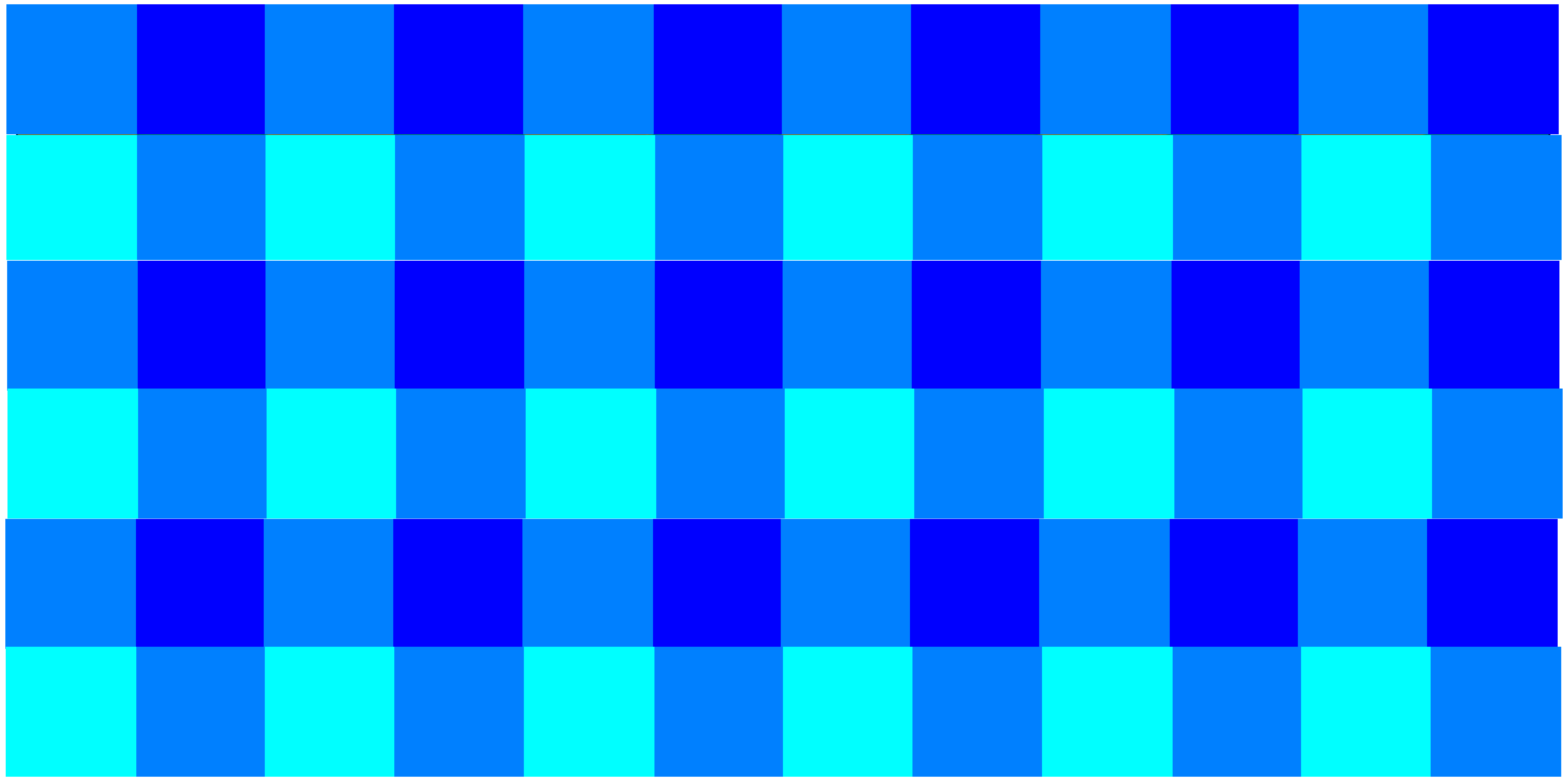
What will demosaiced result look like is this signal was captured by sensor?

Demosaicing errors



(Visualization of signal and Bayer pattern)

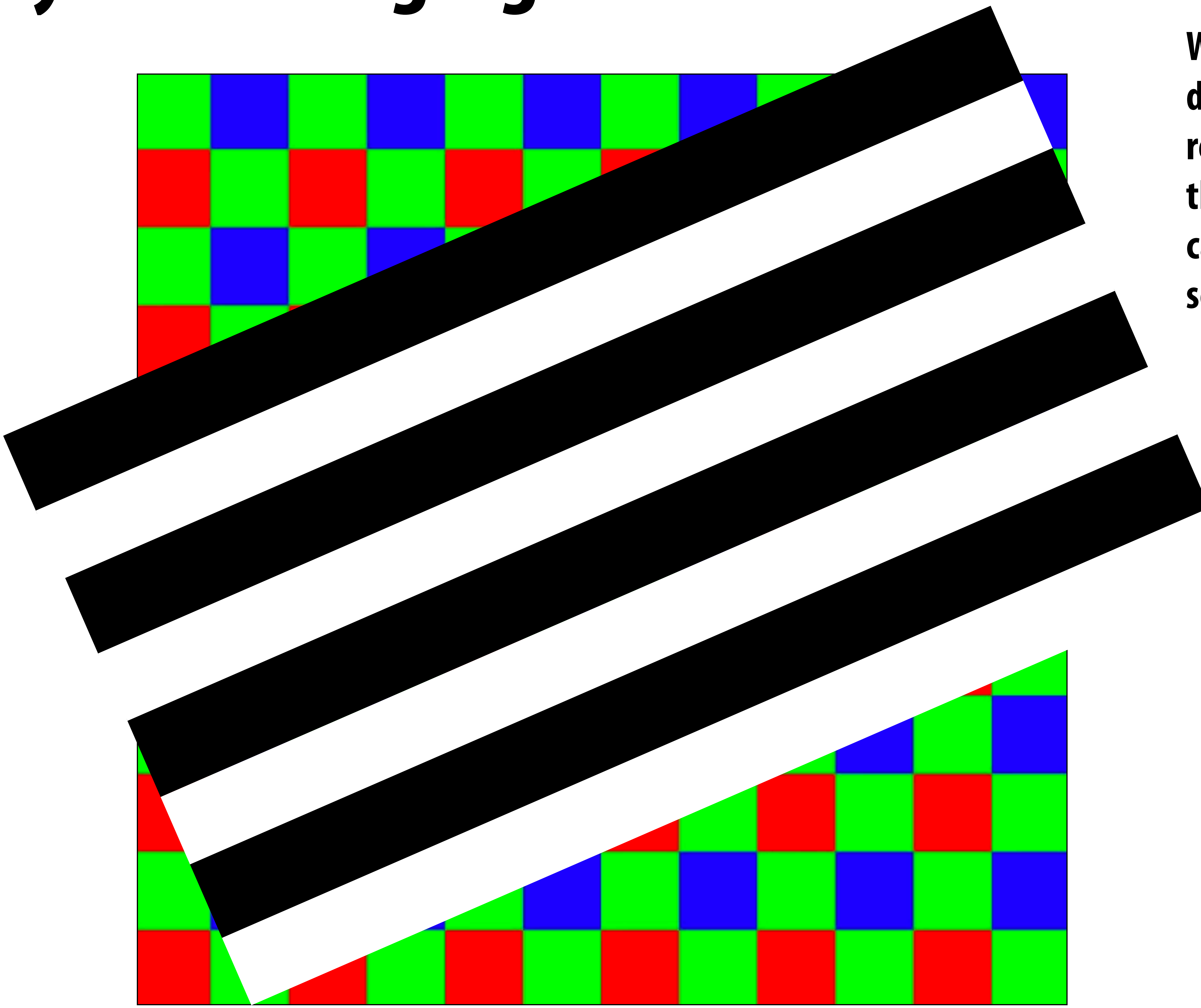
Demosaicing errors



No red measured.

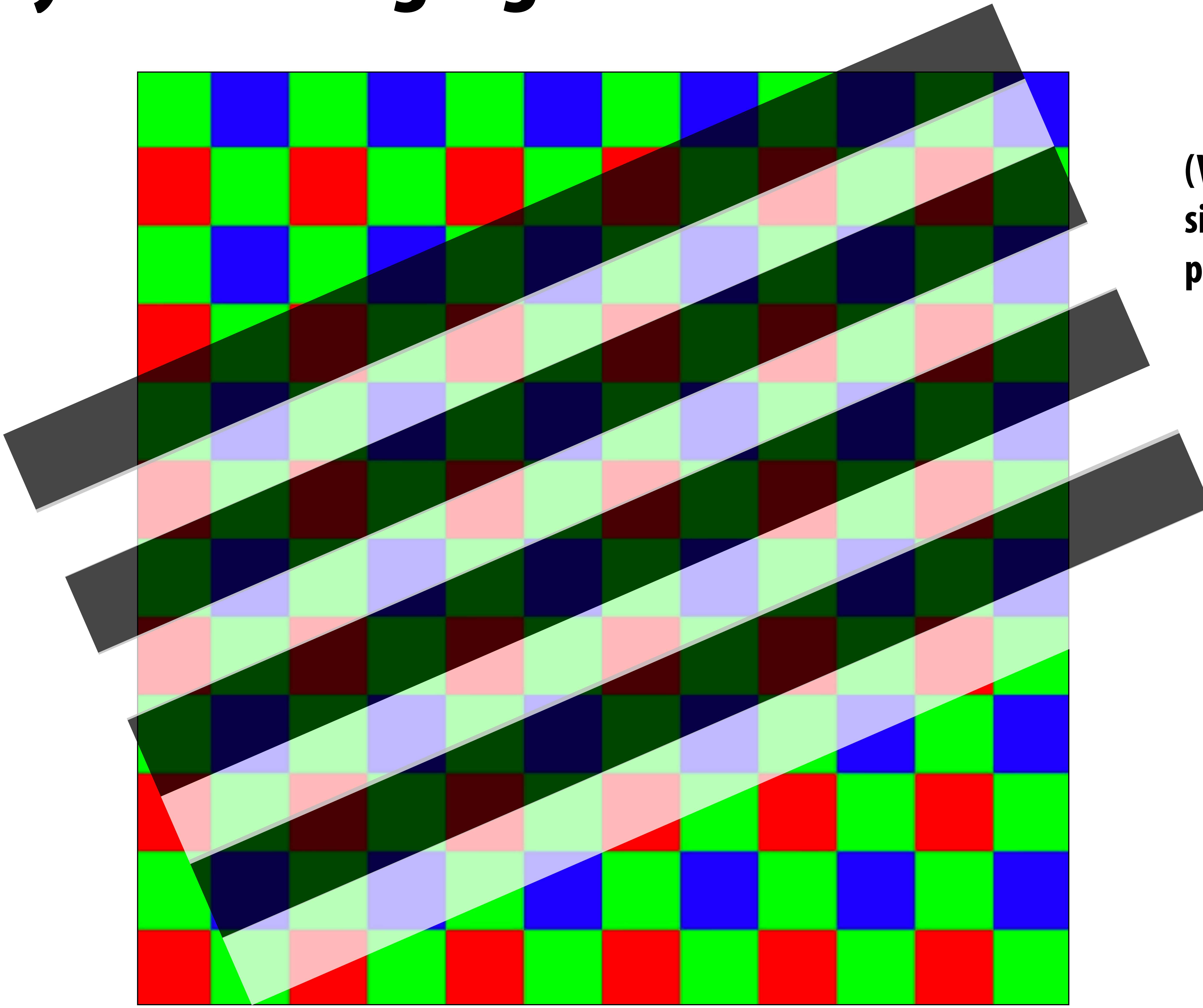
Interpolation of green yields dark/light pattern.

Why color fringing?



What will demosaiced result look like is this signal was captured by sensor?

Why color fringing?



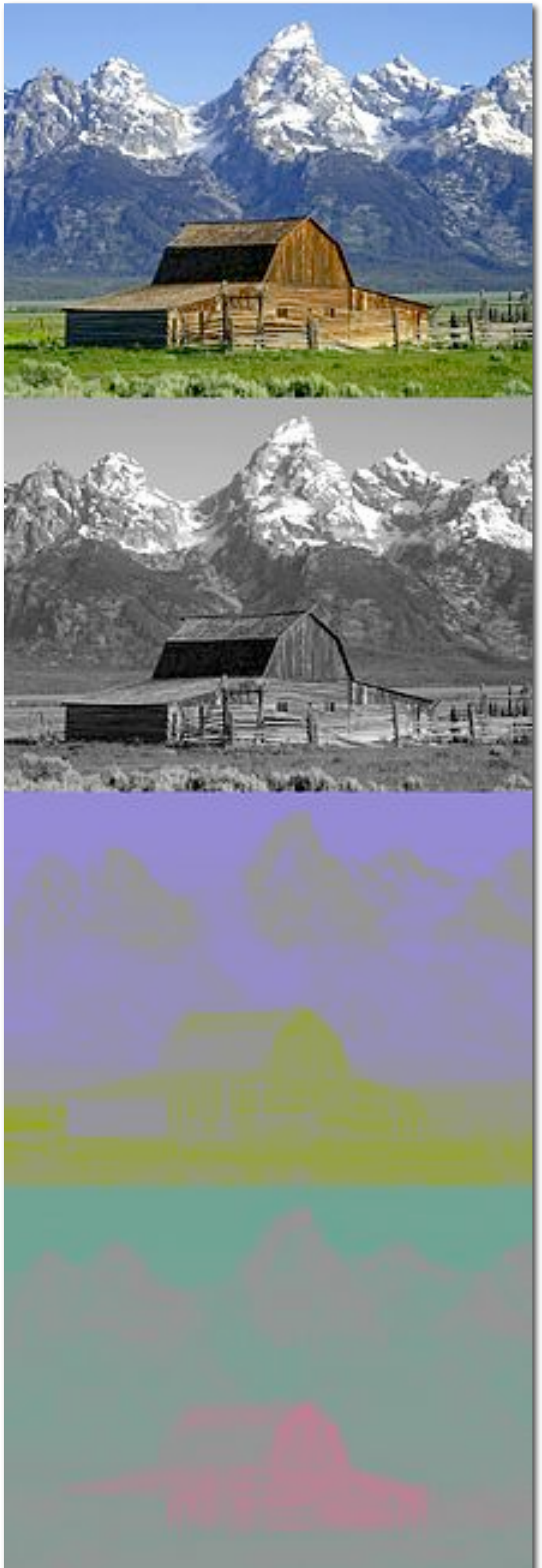
(Visualization of signal and Bayer pattern)

Y'CbCr color space

Recall: colors are represented as point in 3-space

RGB is just one possible basis for representing color

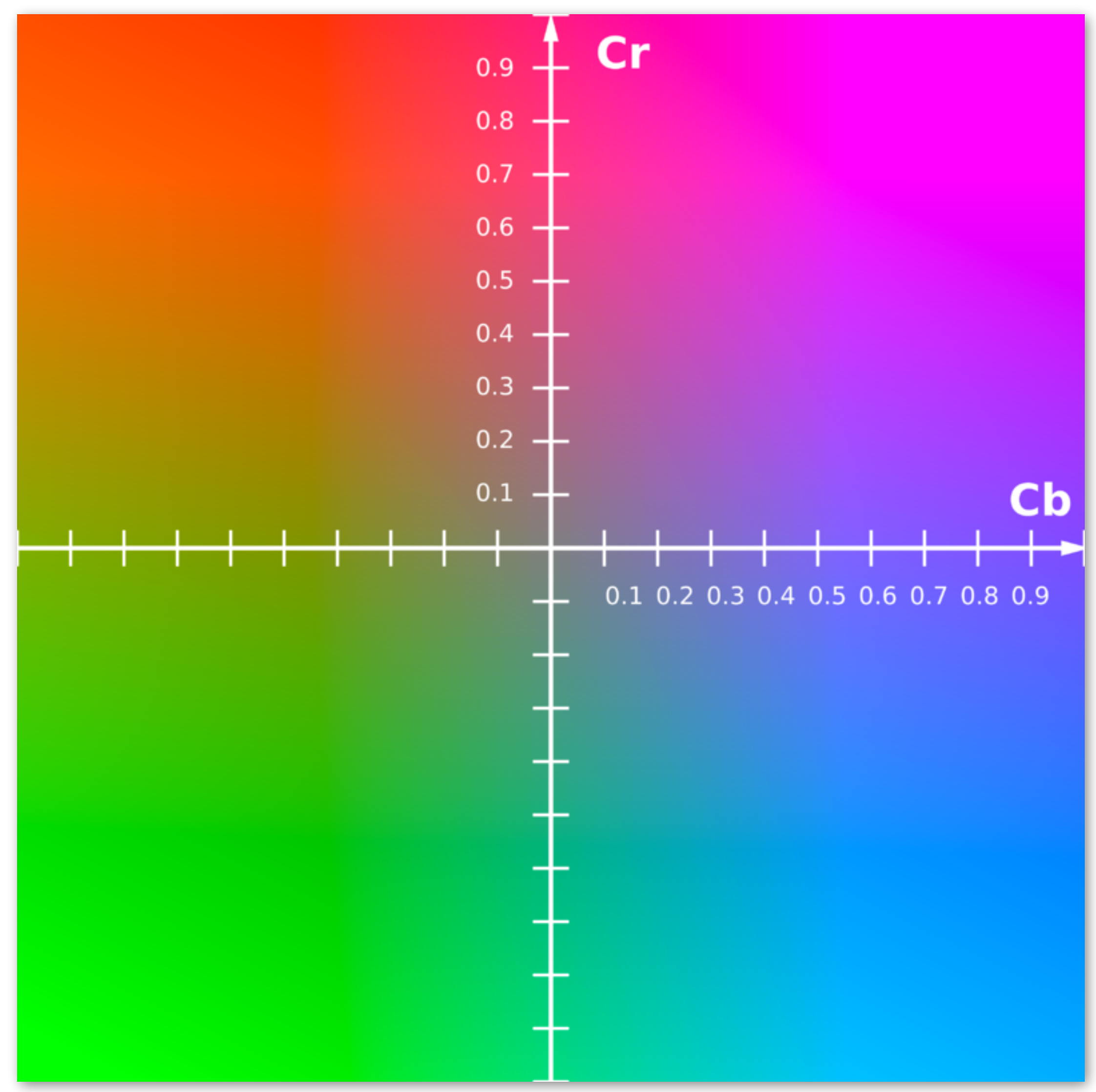
Y'CbCr separates luminance from hue in representation



Y'

Cb

Cr



Y' = luma: perceived luminance

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray

"Gamma corrected" RGB (primed notation indicates perceptual (non-linear) space)

We'll describe what this means this later in the lecture.

Conversion matrix from R'G'B' to Y'CbCr:

$$
\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot G'_D}{256} - \frac{18.285 \cdot B'_D}{256}
\end{aligned}
$$

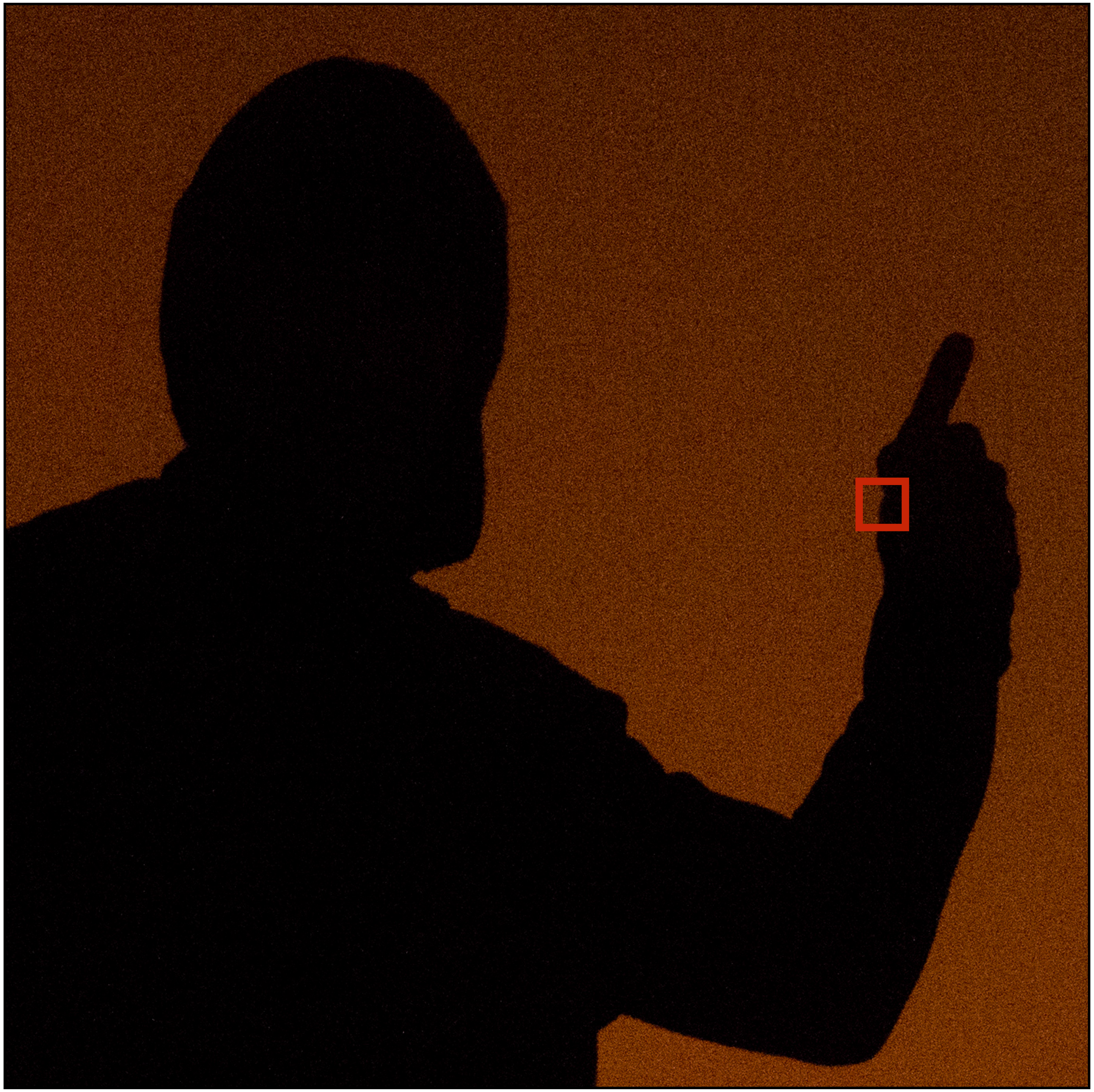
Image credit: Wikipedia

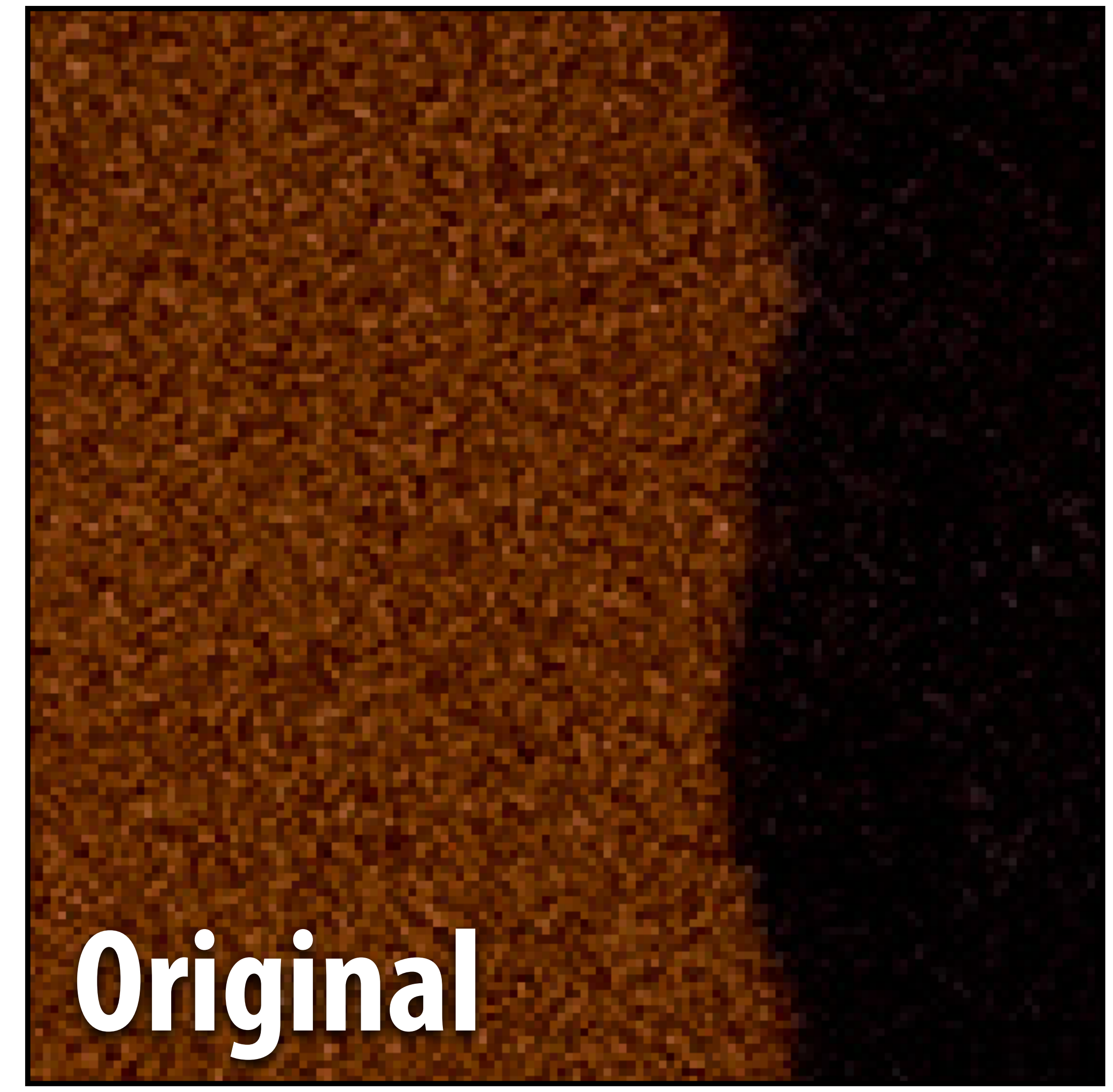
Better demosaic

- Convert demosaiced RGB value to YCbCr
- Low-pass filter (blur) or median filter CbCr channels
- Combine filtered CbCr with full resolution Y from sensor to get RGB

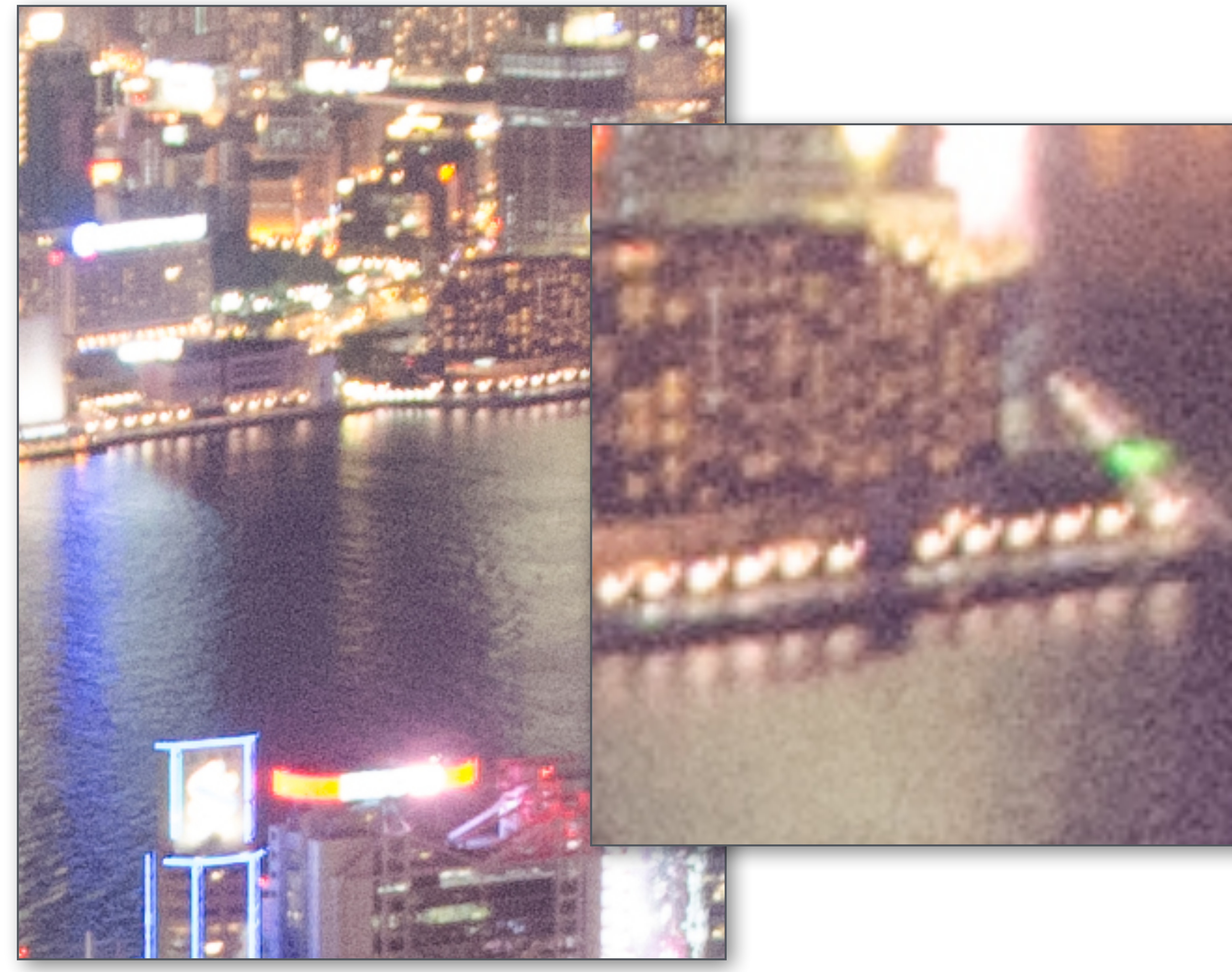
- Trades off spatial resolution of hue to avoid objectionable color fringing

Denoising





Denoising via downsampling



Downsample via point sampling

(noise remains)

Downsample via averaging (bilinear resampling)

Noise reduced

Before talking about denoising…

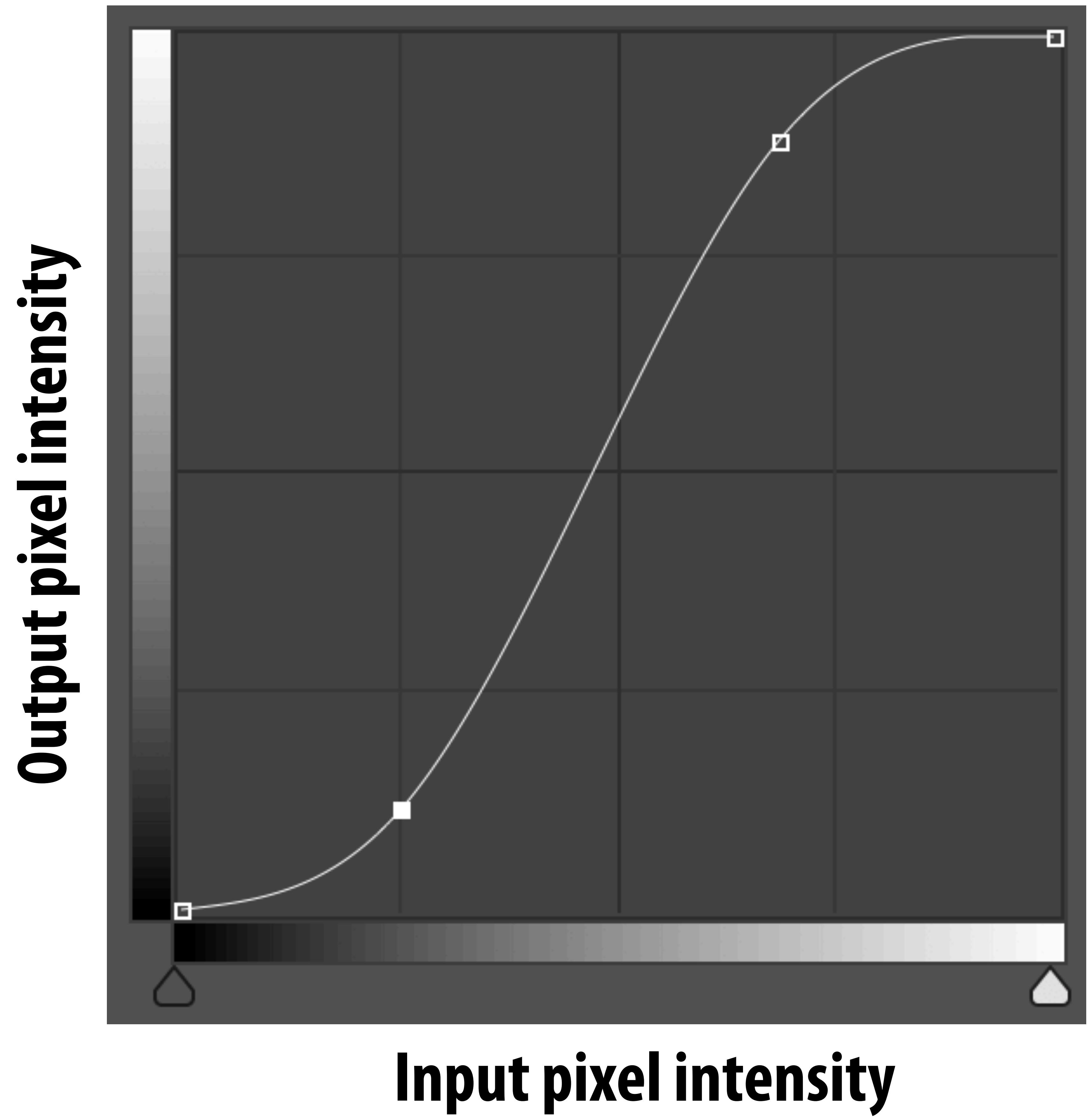
Aside: image processing basics

Example image processing operations

Increase contrast

Increasing contrast with “S curve”

- Per-pixel operation
- output(x,y) = f(input(x,y))



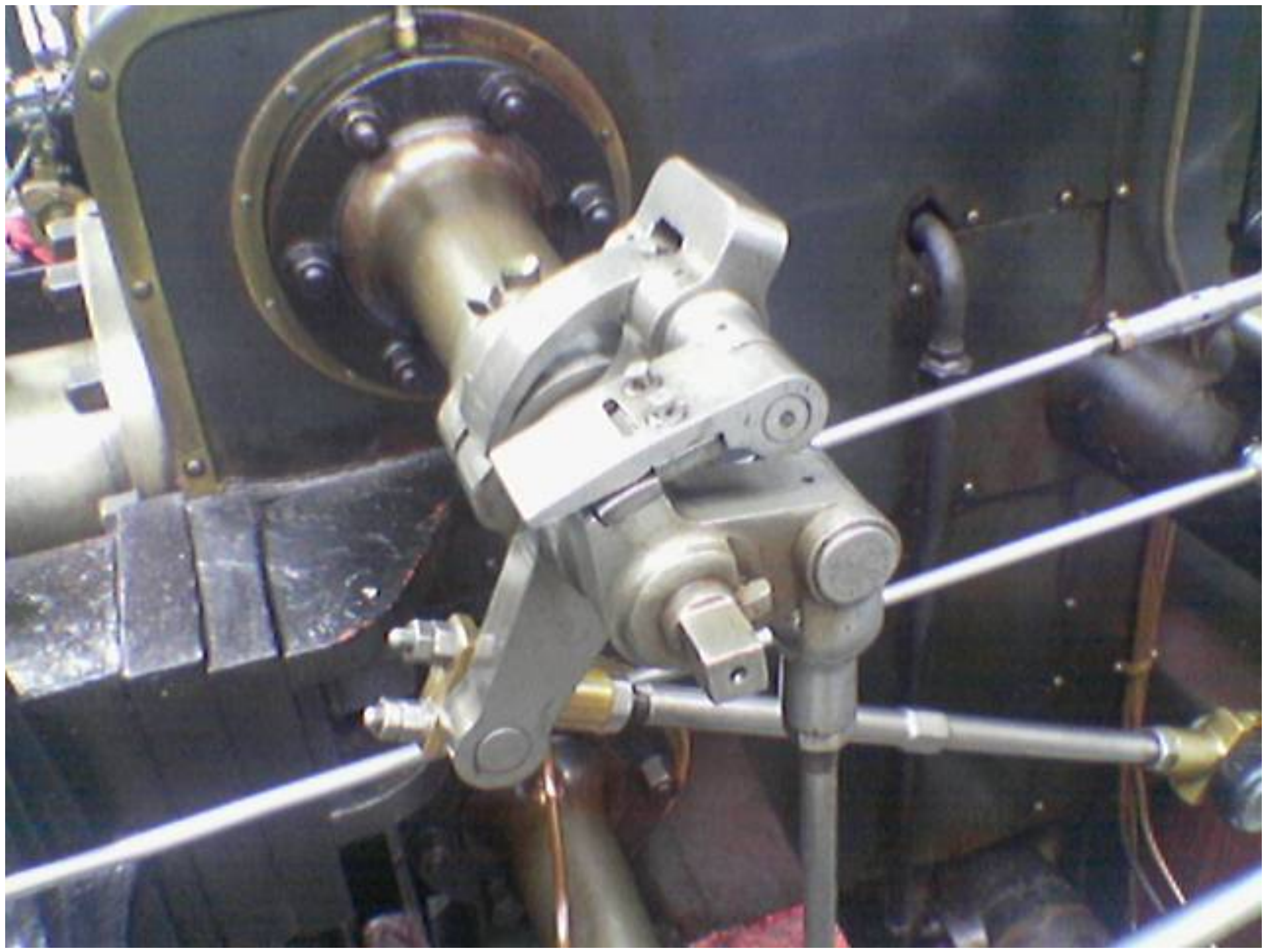
Example image processing operations

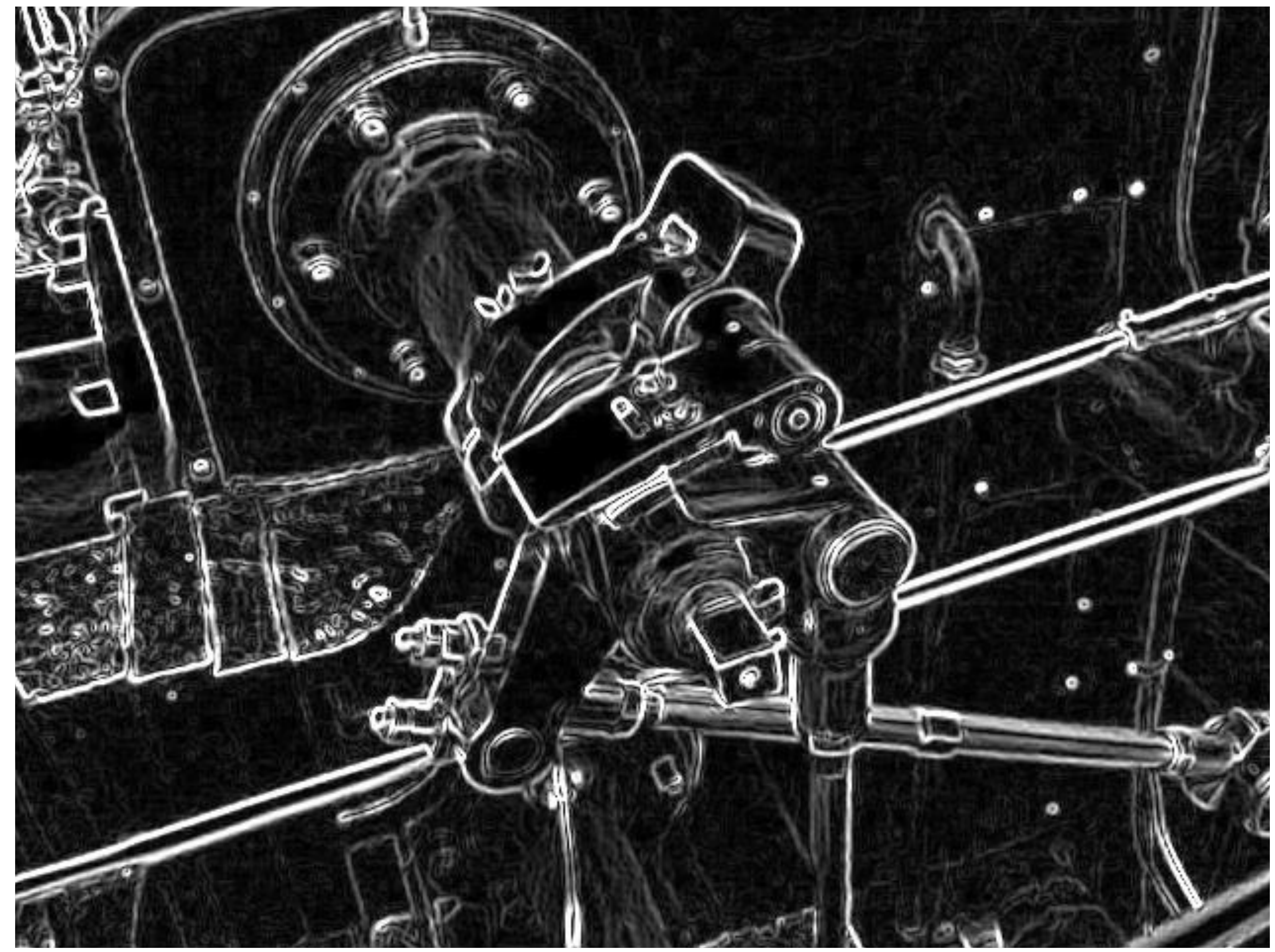
Blur

Example image processing operations

Sharpen

Edge detection





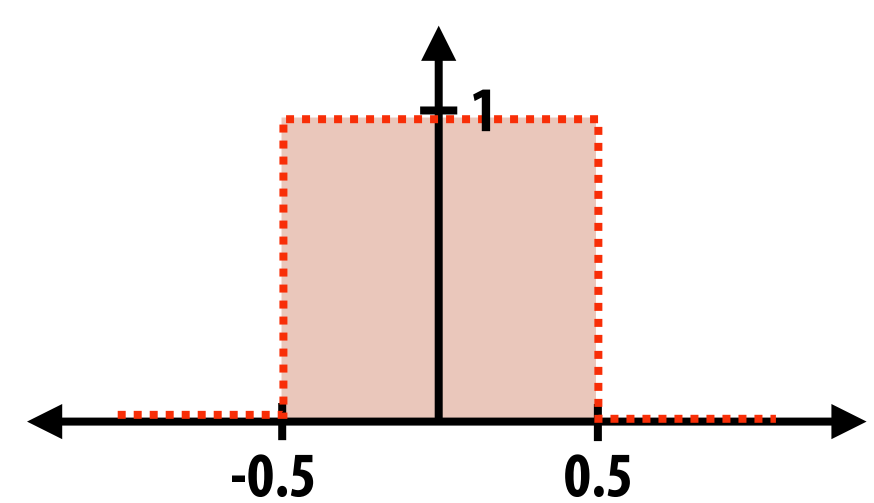
A "smarter" blur (doesn't blur over edges)

Review: convolution

$$\underbrace{(f * g)(x)}_{\text{output signal}} = \int_{-\infty}^{\infty} \underbrace{f(y)}_{\text{filter}}\,\underbrace{g(x - y)}_{\text{input signal}}\,dy$$

It may be helpful to consider the effect of convolution with the simple unit-area "box" function:

$$f(x) = \begin{cases} 1 & |x| \leq 0.5 \\ 0 & otherwise \end{cases}$$



$$(f * g)(x) = \int_{-0.5}^{0.5} g(x - y)dy$$

f * g* is a "blurred" version of *g

Discrete 2D convolution

$$\underbrace{(f*g)(x,y)}_{\text{output image}} = \sum_{i,j=-\infty}^{\infty} \underbrace{f(i,j)}_{\text{filter}}\,\underbrace{I(x-i,y-j)}_{\text{input image}}$$

Consider $f(i,j)$ **that is nonzero only when:** $-1 \leq i,j \leq 1$

Then:

$$(f*g)(x,y) = \sum_{i,j=-1}^{1} f(i,j)I(x-i,y-j)$$

And we can represent f(i,j) as a 3x3 matrix of values where:

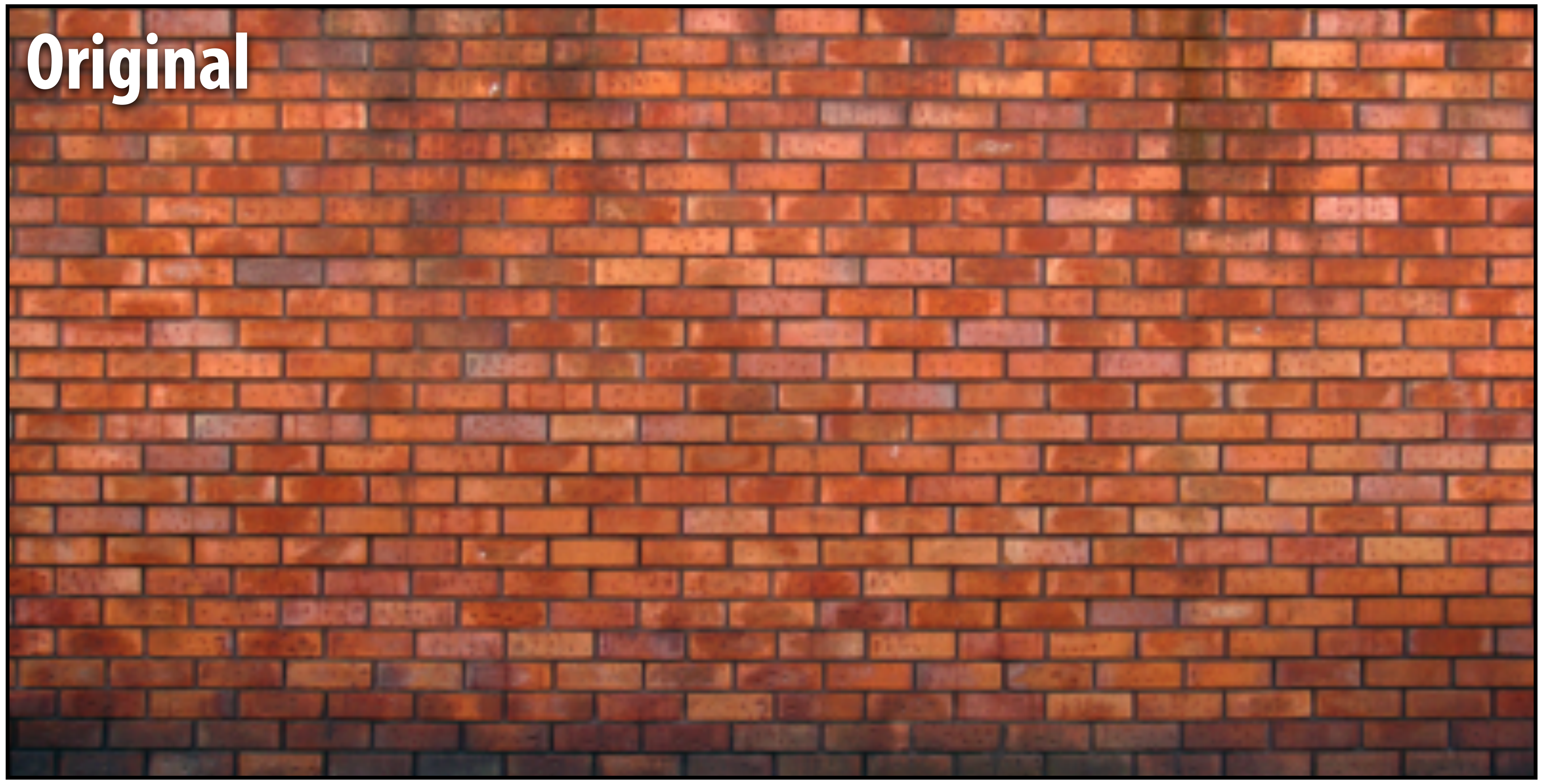
$f(i,j) = \mathbf{F}_{i,j}$ **(often called: "filter weights", "filter kernel")**

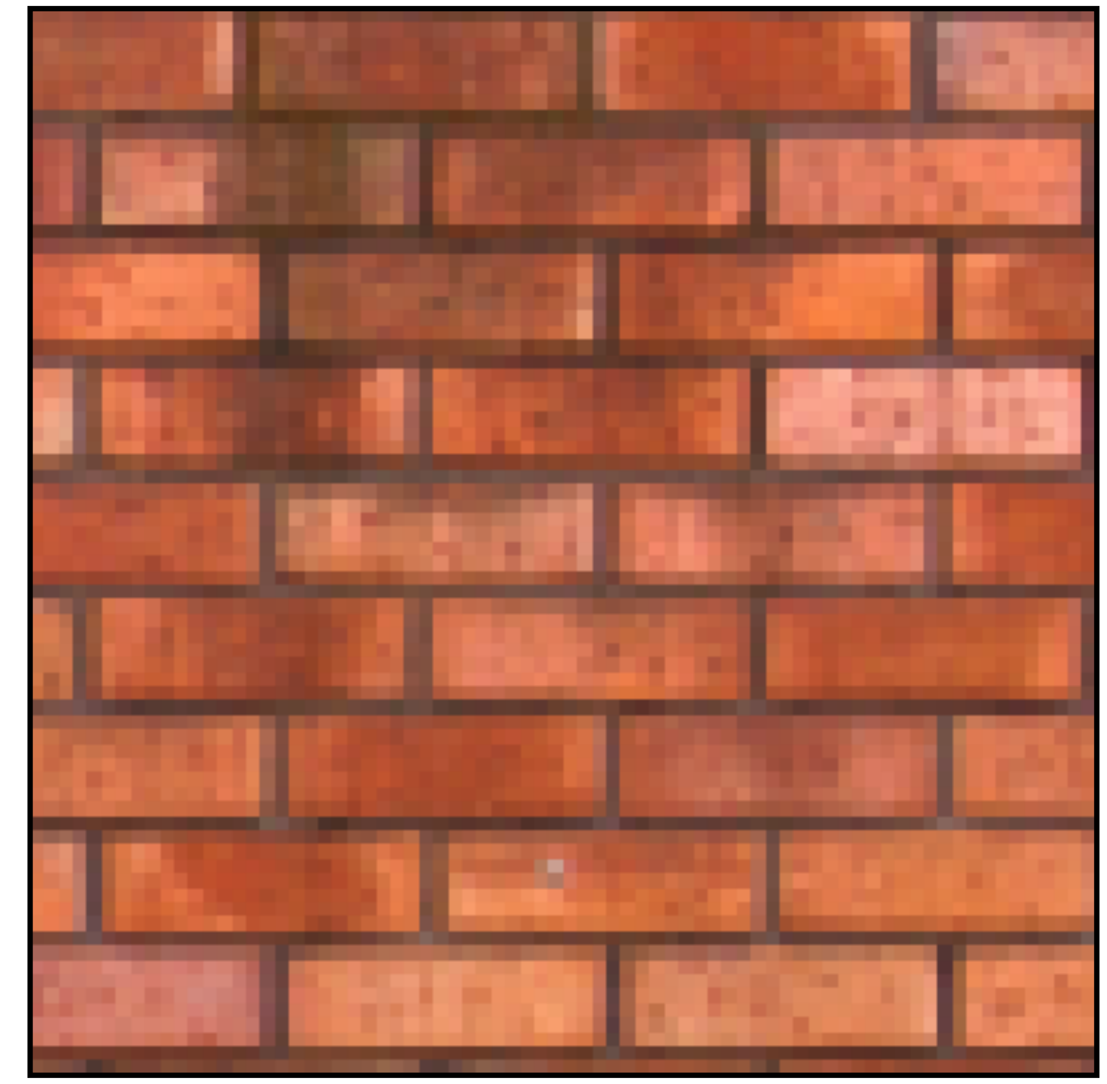
Simple 3x3 box blur in code

```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9,
                   1./9, 1./9, 1./9};


for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<3; jj++)
         for (int ii=0; ii<3; ii++)
            tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH + i] = tmp;
  }
}
```

For now: ignore boundary pixels and assume output image is smaller than input (makes convolution loop bounds much simpler to write)

7x7 box blur





Gaussian blur

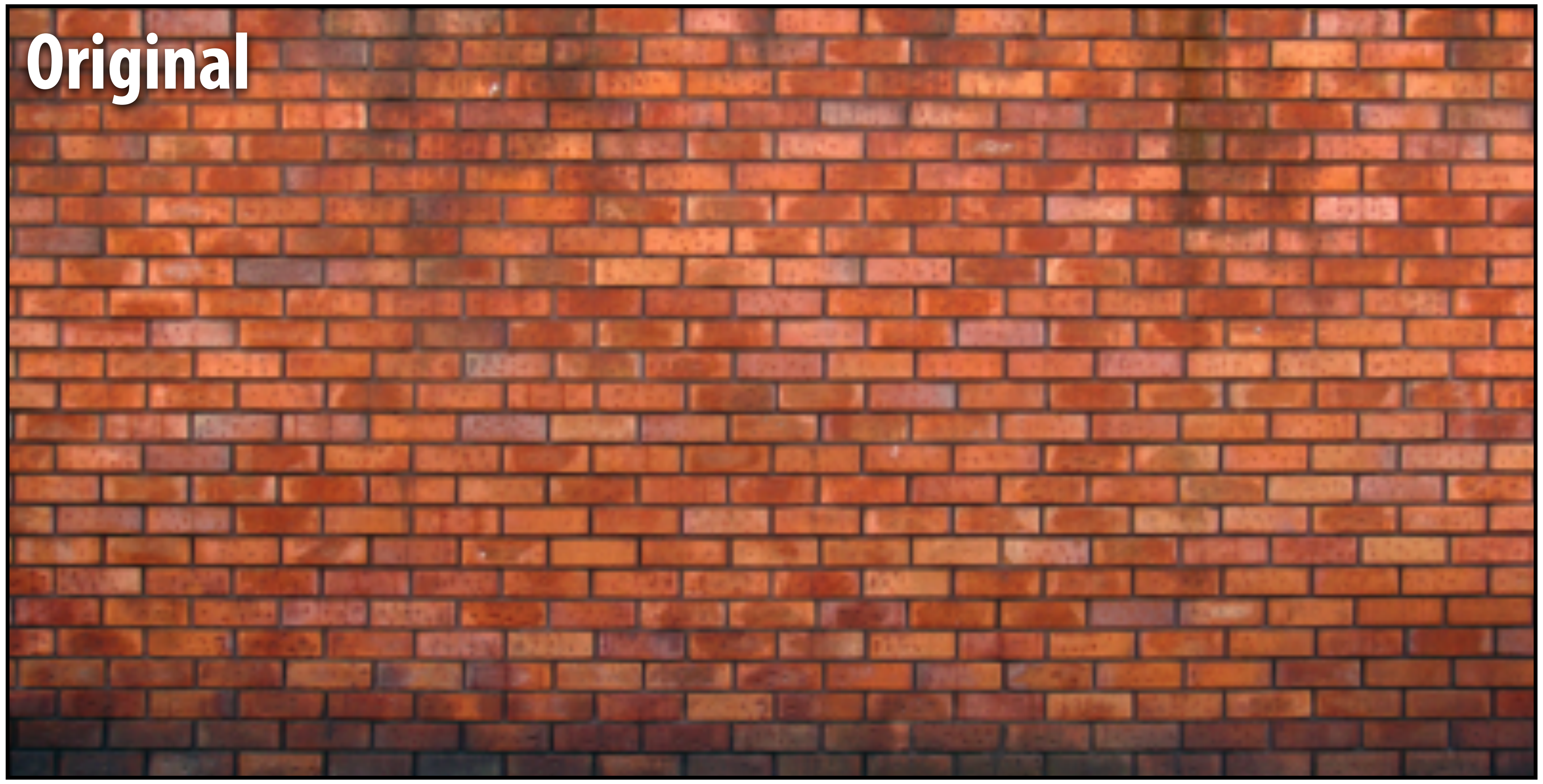
- **Obtain filter coefficients from sampling 2D Gaussian**

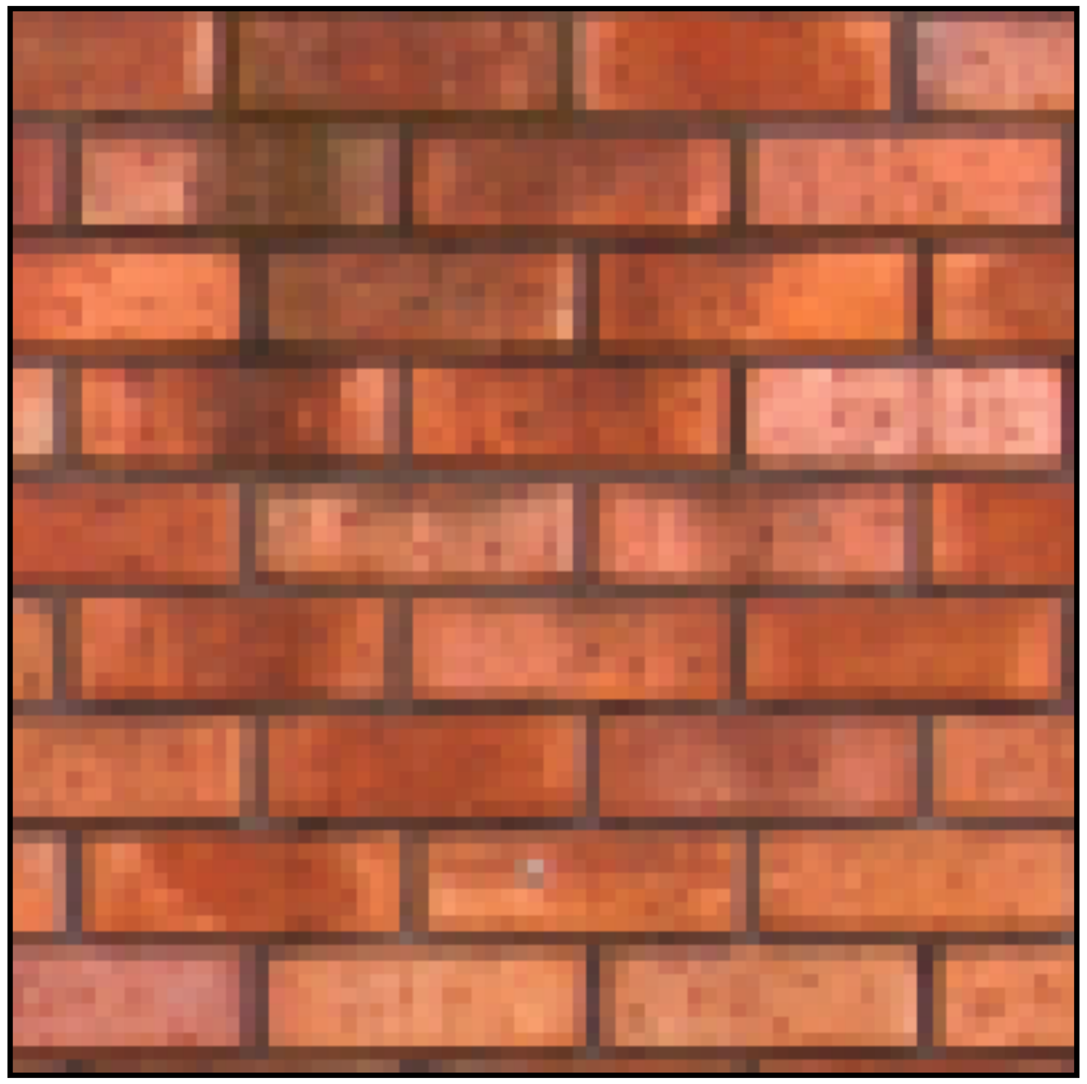
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

$$\begin{bmatrix} .075 & .124 & .075 \\ .124 & .204 & .124 \\ .075 & .124 & .075 \end{bmatrix}$$

7x7 gaussian blur



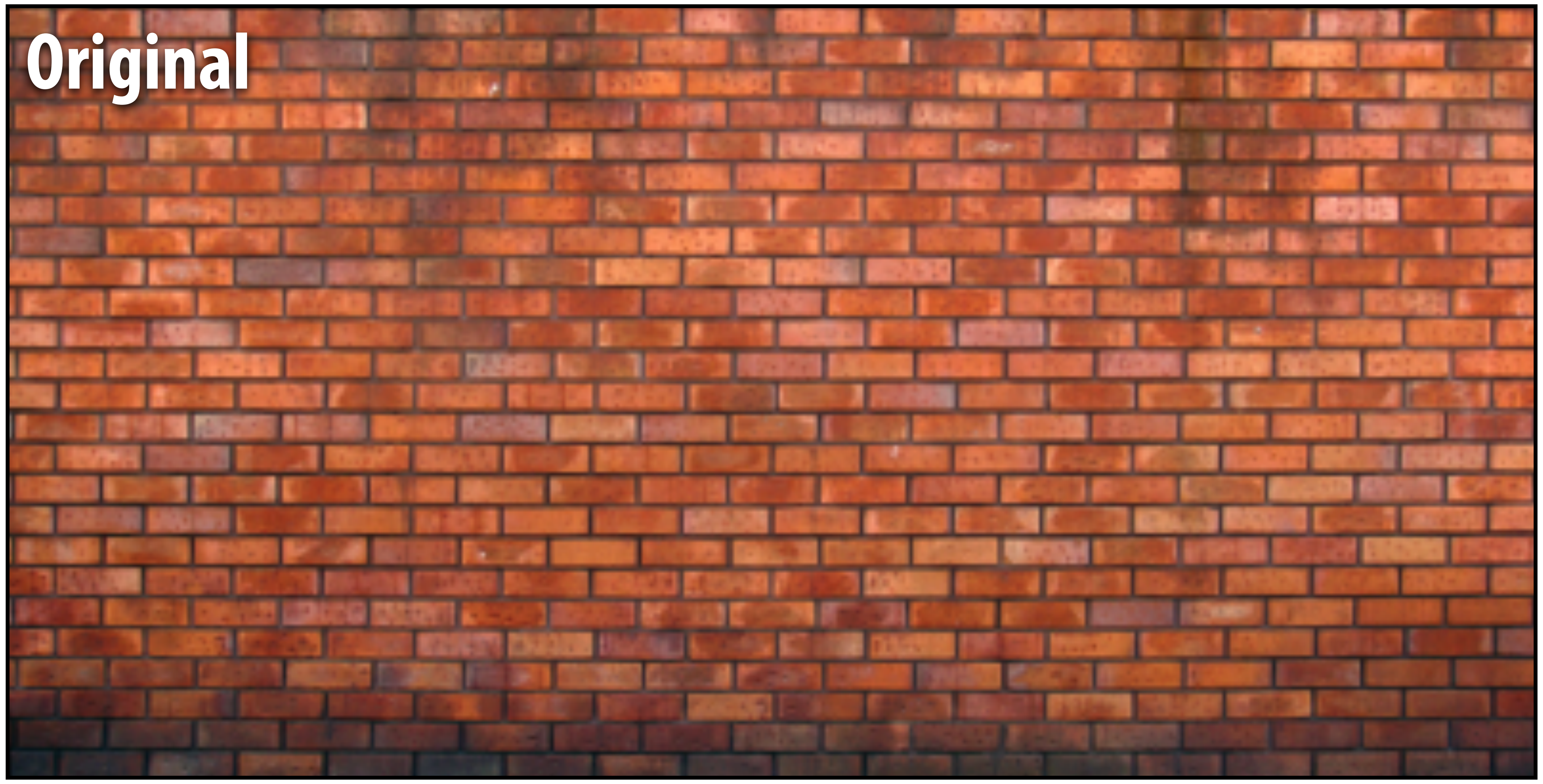


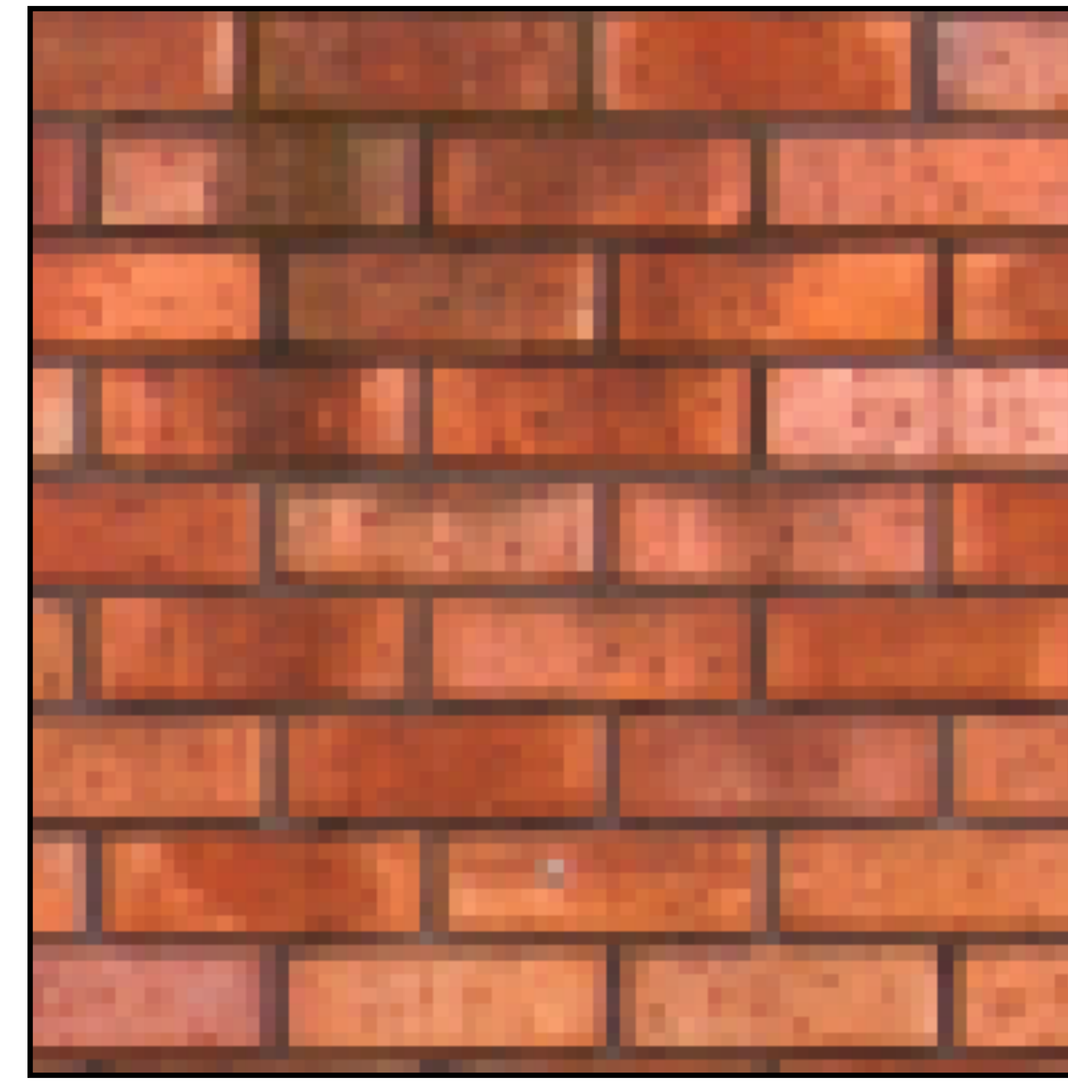
What does convolution with this filter do?

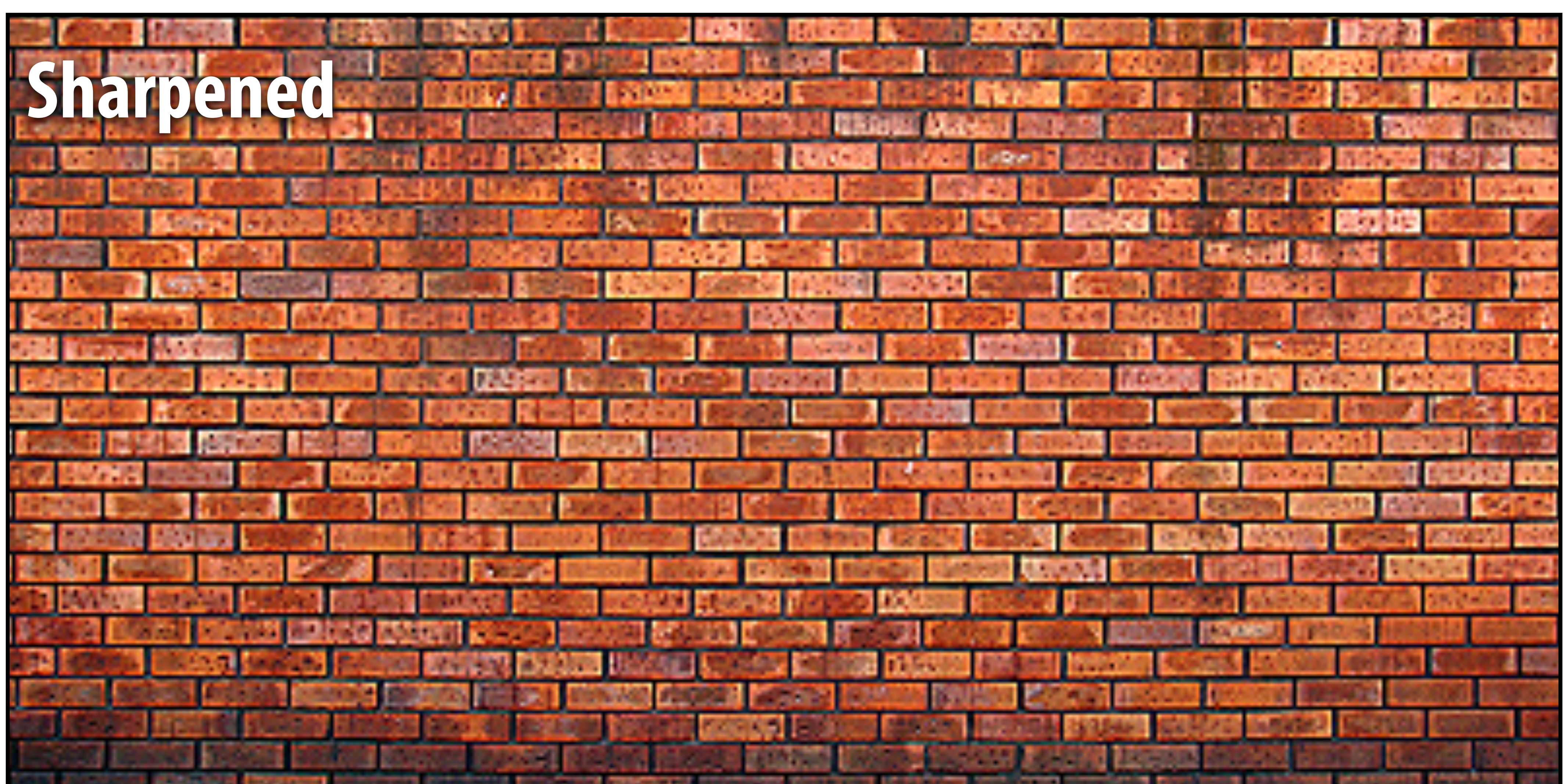
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

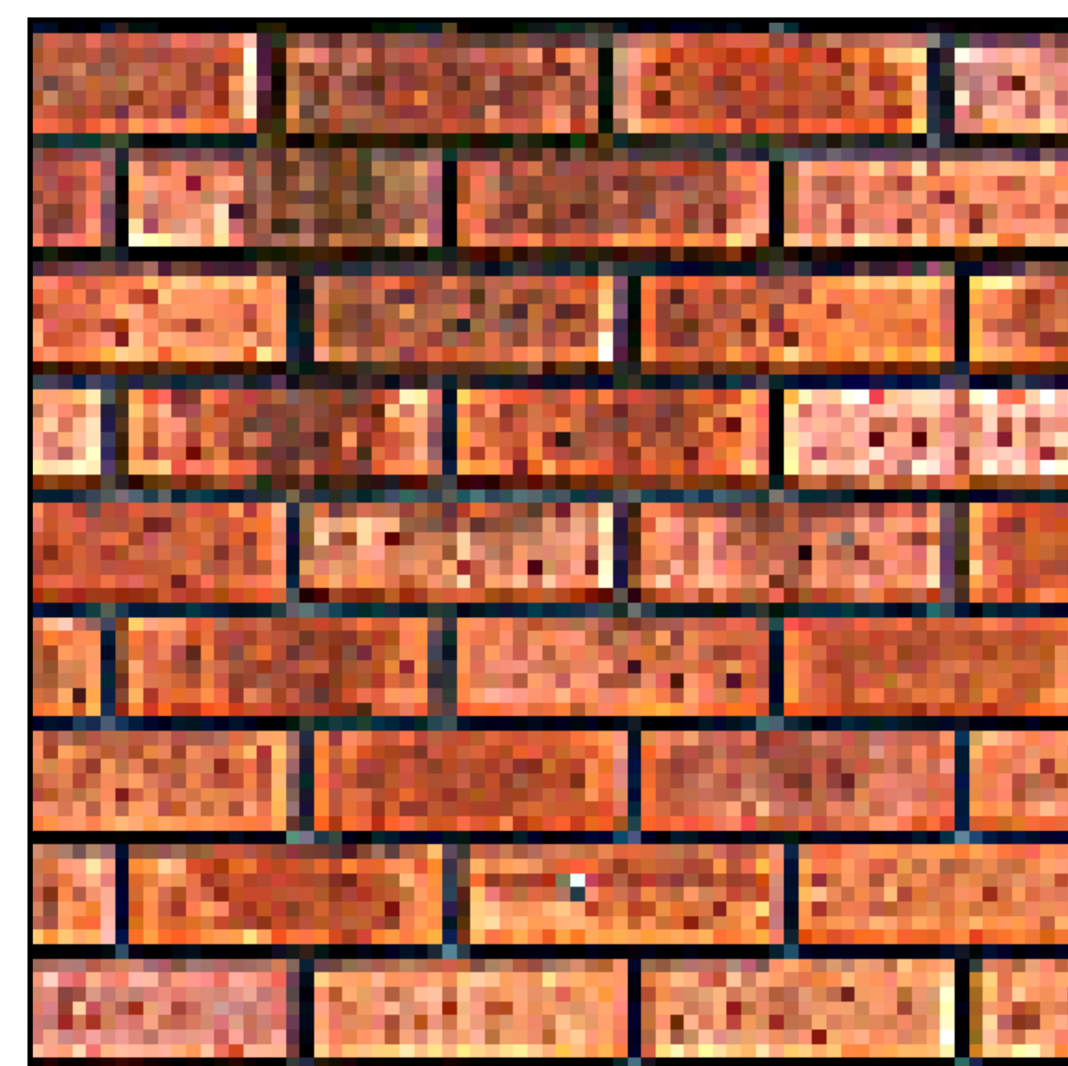
Sharpens image!

3x3 sharpen filter









What does convolution with these filters do?

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

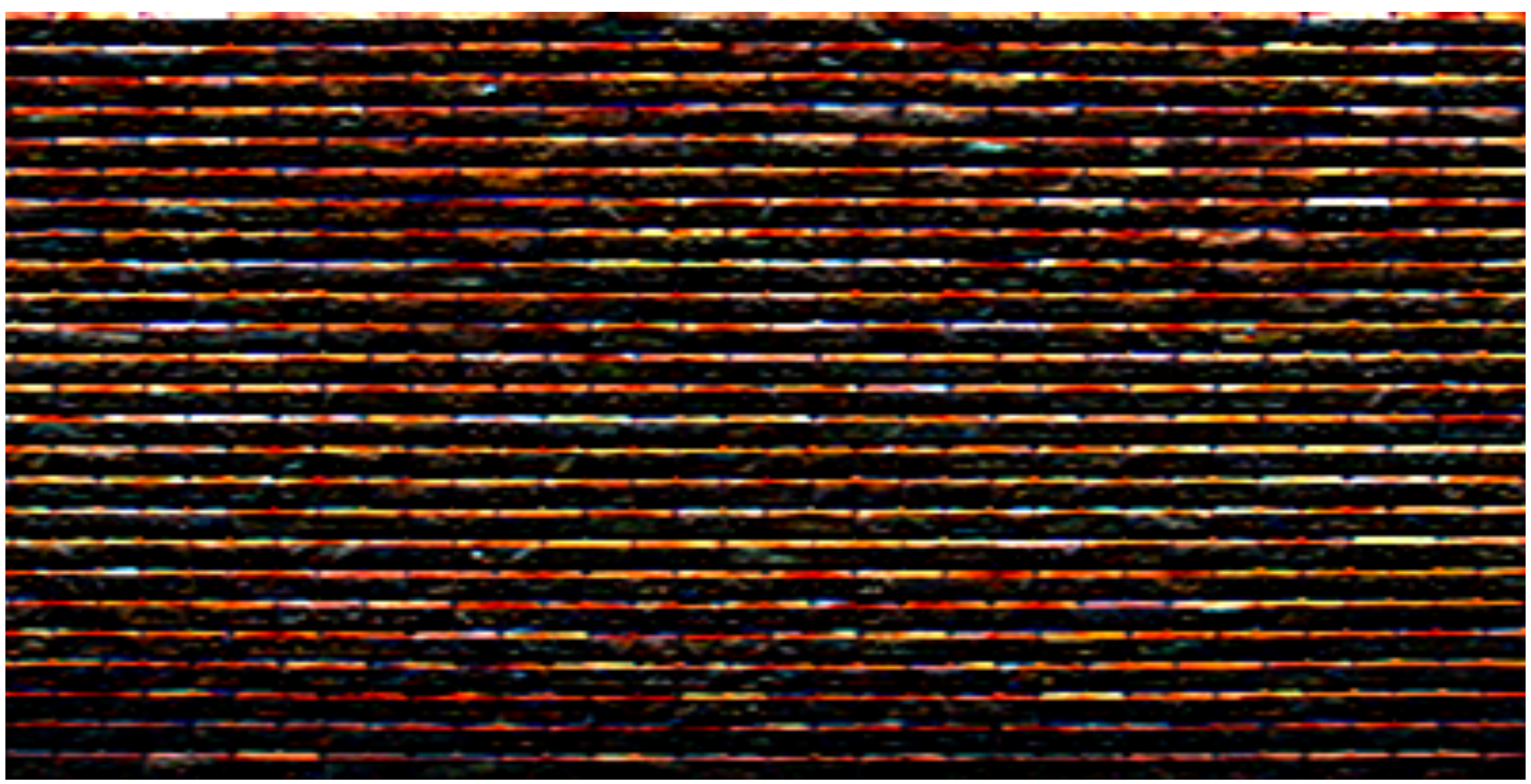
Extracts horizontal gradients

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Extracts vertical gradients

Gradient detection filters

Horizontal gradients



Vertical gradients

Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image (this is a common interpretation in computer vision)

Sobel edge detection

- **Compute gradient response images**

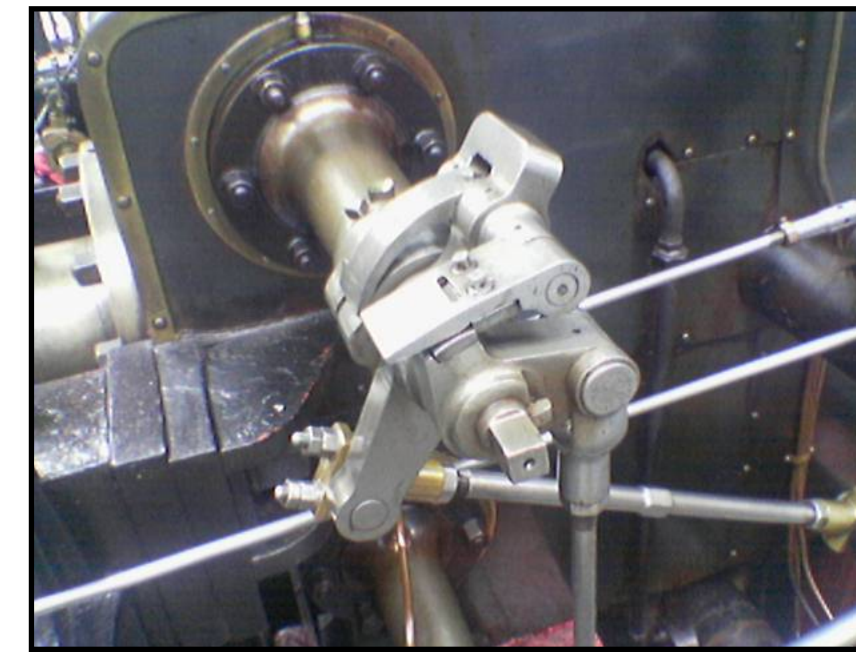
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

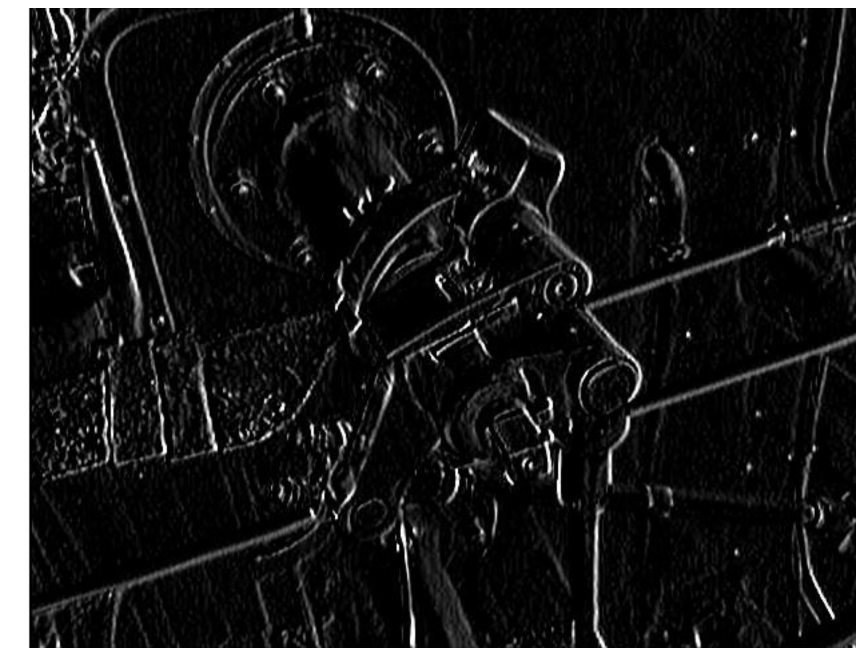
- **Find pixels with large gradients**

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

Pixel-wise operation on images

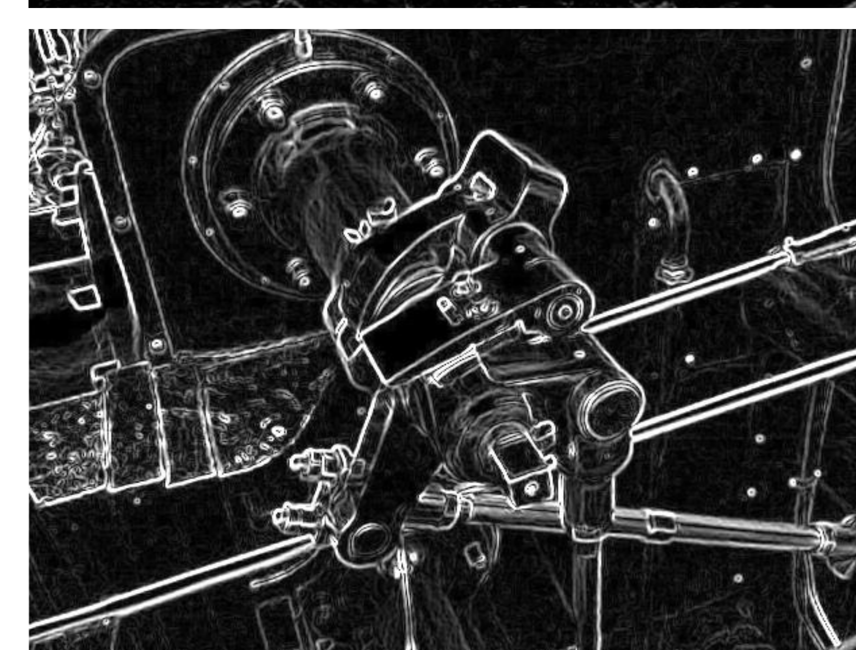


G_x



G_y

G



Data-dependent filter (not a convolution)

```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      float min_value = min( min(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                             min(input[j*WIDTH + i-1], input[j*WIDTH + i+1]) );
      float max_value = max( max(input[(j-1)*WIDTH + i], input[(j+1)*WIDTH + i]),
                             max(input[j*WIDTH + i-1], input[j*WIDTH + i+1]) );
      output[j*WIDTH + i] = clamp(min_value, max_value, input[j*WIDTH + i]);
    }
}
```

This filter clamps pixels to the min/max of its cardinal neighbors (e.g., hot-pixel suppression — no need for a lookup table)

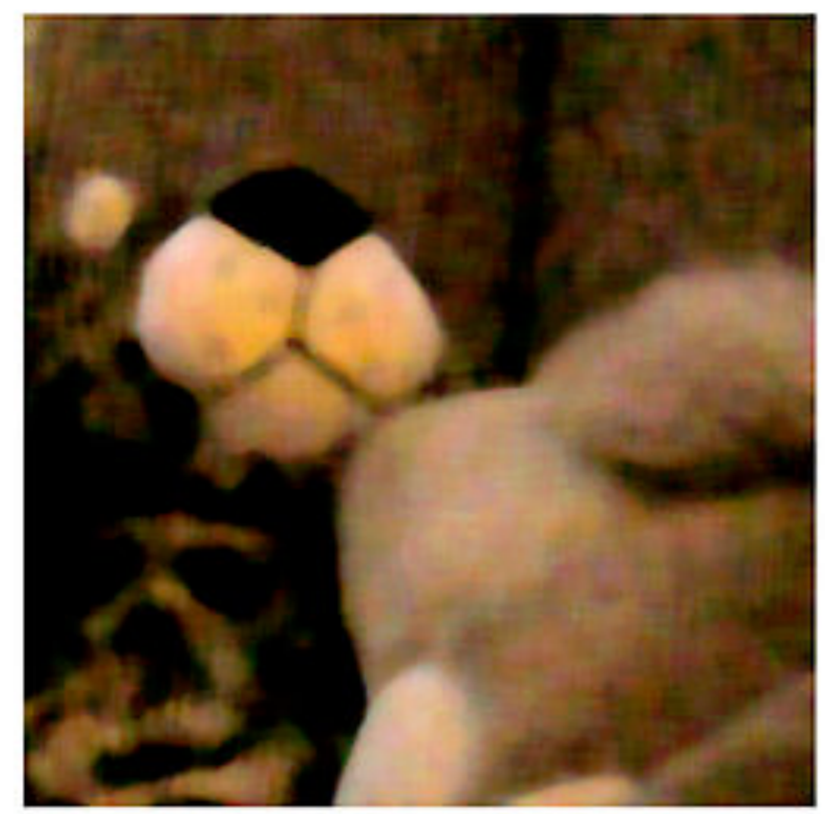
Median filter

- **Replace pixel with median of its neighbors**
  - Useful noise reduction filter: unlike gaussian blur, one bright pixel doesn't drag up the average for entire region
- **Not linear, not separable**
  - Filter weights are 1 or 0 (depending on image content)

```
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];
for (int j=0; j<HEIGHT; j++) {
   for (int i=0; i<WIDTH; i++) {
      output[j*WIDTH + i] =
           // compute median of pixels
           // in surrounding 5x5 pixel window
   }
}
```

original image

1px median filter



3px median filter

10px median filter

- **Basic algorithm for NxN support region:**
  - Sort N^2 elements in support region, then pick median: $O(N^2 \log(N^2))$ work per pixel
  - Can you think of an $O(N^2)$ algorithm? What about $O(N)$?

5x5 median filter (N=5)

- $O(N^2)$ work-per-pixel solution for 8-bit pixel data (radix sort 8 bit-integer data)
  - Bin all pixels in support region, then scan histogram bins to find median

```
int WIDTH = 1024;
int HEIGHT = 1024;
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];
int histogram[256];

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    // construct histogram of support region
    for (int ii=0; ii<256; ii++)
      histogram[ii] = 0;
    for (int jj=0; jj<5; jj++)
      for (int ii=0; ii<5; ii++)
         histogram[input[(j+jj)*(WIDTH+2) + (i+ii)]]++;

    // scan the 256 bins to find median
    // median value of 5x5=25 elements is bin containing 13th value
    int count = 0;
    for (int ii=0; ii<256; i++) {
       if (count + histogram[ii] >= 13)
         output[j*WIDTH + i] = uint8(ii);
       count += histogram[ii];
    }
  }
}
```

See Weiss [SIGGRAPH 2006] for O(lg N) work-per-pixel median filter (incrementally updates histogram)

Bilateral filter

Example use of bilateral filter: removing noise while preserving image edges

Bilateral filter

$$\mathrm{BF}[I](p) = \underbrace{\frac{1}{W_p}}_{\text{Normalization}} \underbrace{\sum_{i,j}}_{\substack{\text{For all pixels in support region}\\ \text{of Gaussian kernel}}} f(|\underbrace{I(x-i,y-j) - I(x,y)}_{\substack{\text{Re-weight based on difference}\\ \text{in input image pixel values}}}|)\overbrace{G_\sigma(i,j)}^{\text{Gaussian blur kernel}}\overbrace{I(x-i,y-j)}^{\text{Input image}}$$

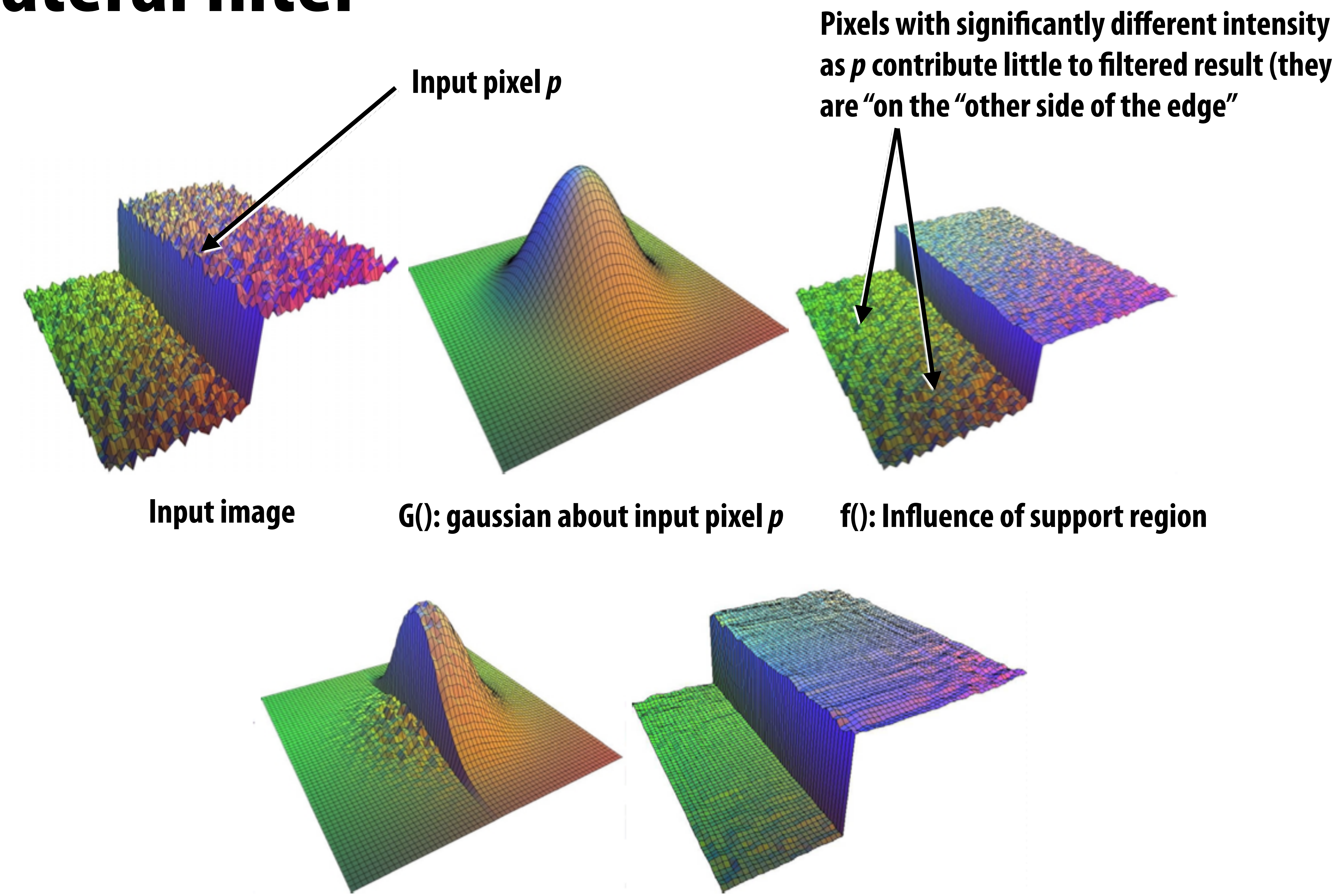
$$W_p = \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|)G_\sigma(i,j)I(x-i,y-j)$$

- **The bilateral filter is an "edge preserving" filter: down-weight contribution of pixels on the "other side" of strong edges. $f(x)$ defines what "strong edge means"**
- **Spatial distance weight term $f(x)$ could itself be a gaussian**
  - **Or very simple:** $f(x) = 0 \text{ if } x > \mathit{threshold},\ 1\ \mathit{otherwise}$

Value of output pixel (x,y) is the weighted sum of all pixels in the support region of a truncated gaussian kernel

But weight is combination of <u>spatial distance</u> and <u>input image pixel intensity</u> difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

Bilateral filter



Input image

G(): gaussian about input pixel *p*

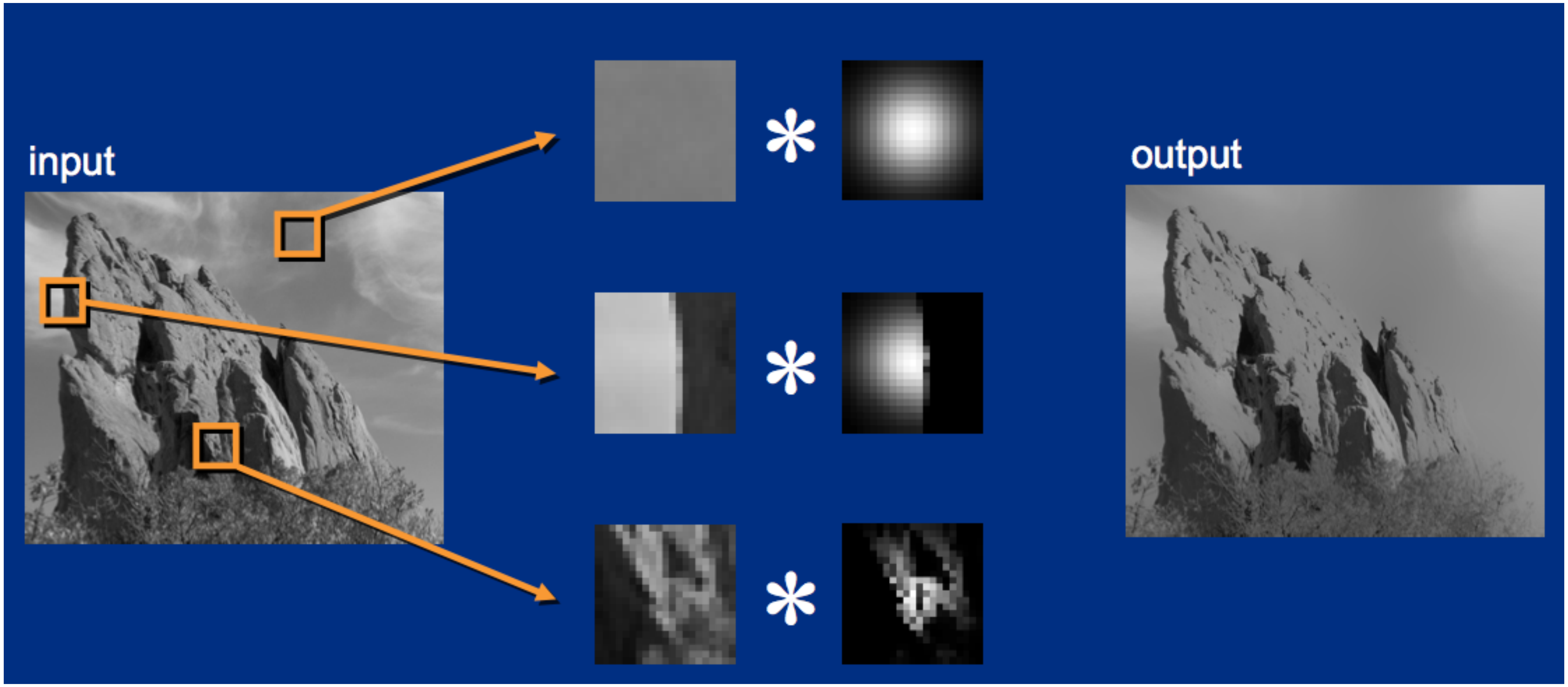
f(): Influence of support region

G x f: filter weights for pixel *p*

Filtered output image

Figure credit: Durand and Dorsey, “Fast Bilateral Filtering for the Display of High-Dynamic-Range Images”, SIGGRAPH 2002

Bilateral filter: kernel depends on image content



See Paris et al. [ECCV 2006] for a fast approximation to the bilateral filter

Question: describe a type of edge the bilateral filter will not respect (it will blur across these edges)

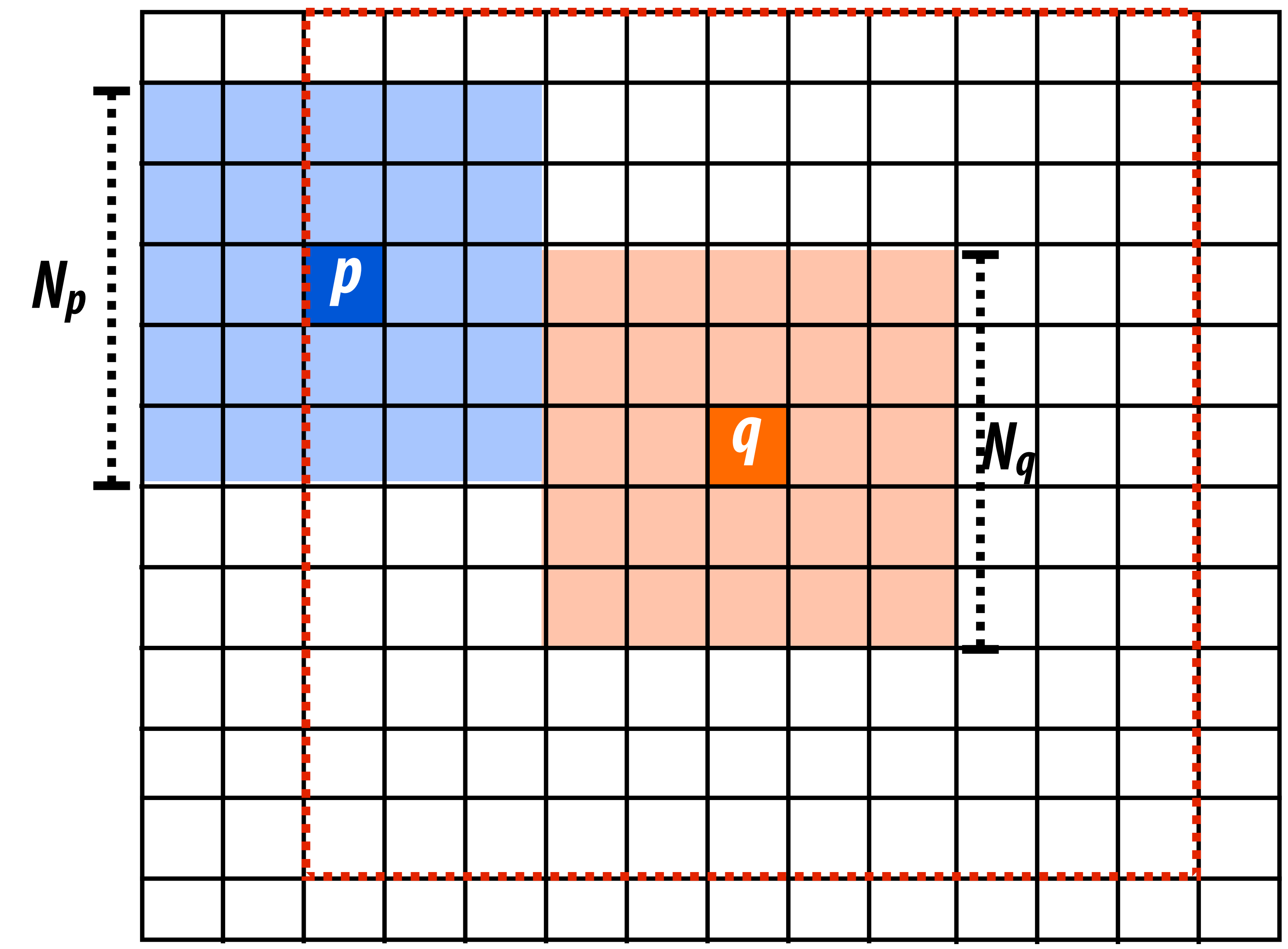
Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

Denoising using non-local means

- Main assumption: images have repeating texture
- Main idea: replace pixel with average value of nearby pixels that have a similar surrounding region

$$\mathrm{NL}[I](p) = \sum_{q \in S} w(p,q) I(q)$$

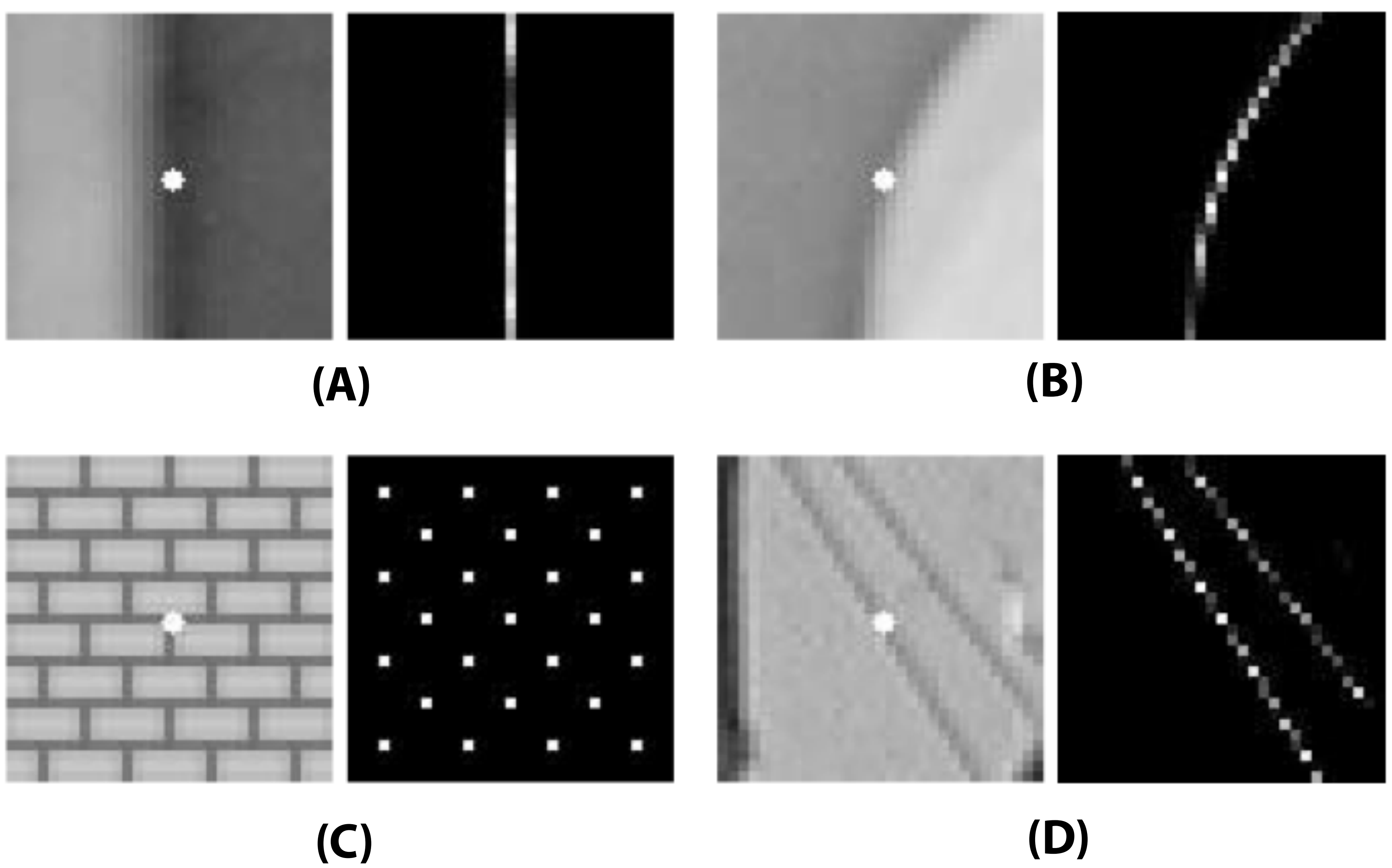
$$w(p,q) = \frac{1}{C_p} e^{\frac{-\| N_p - N_q \|^2}{h^2}}$$



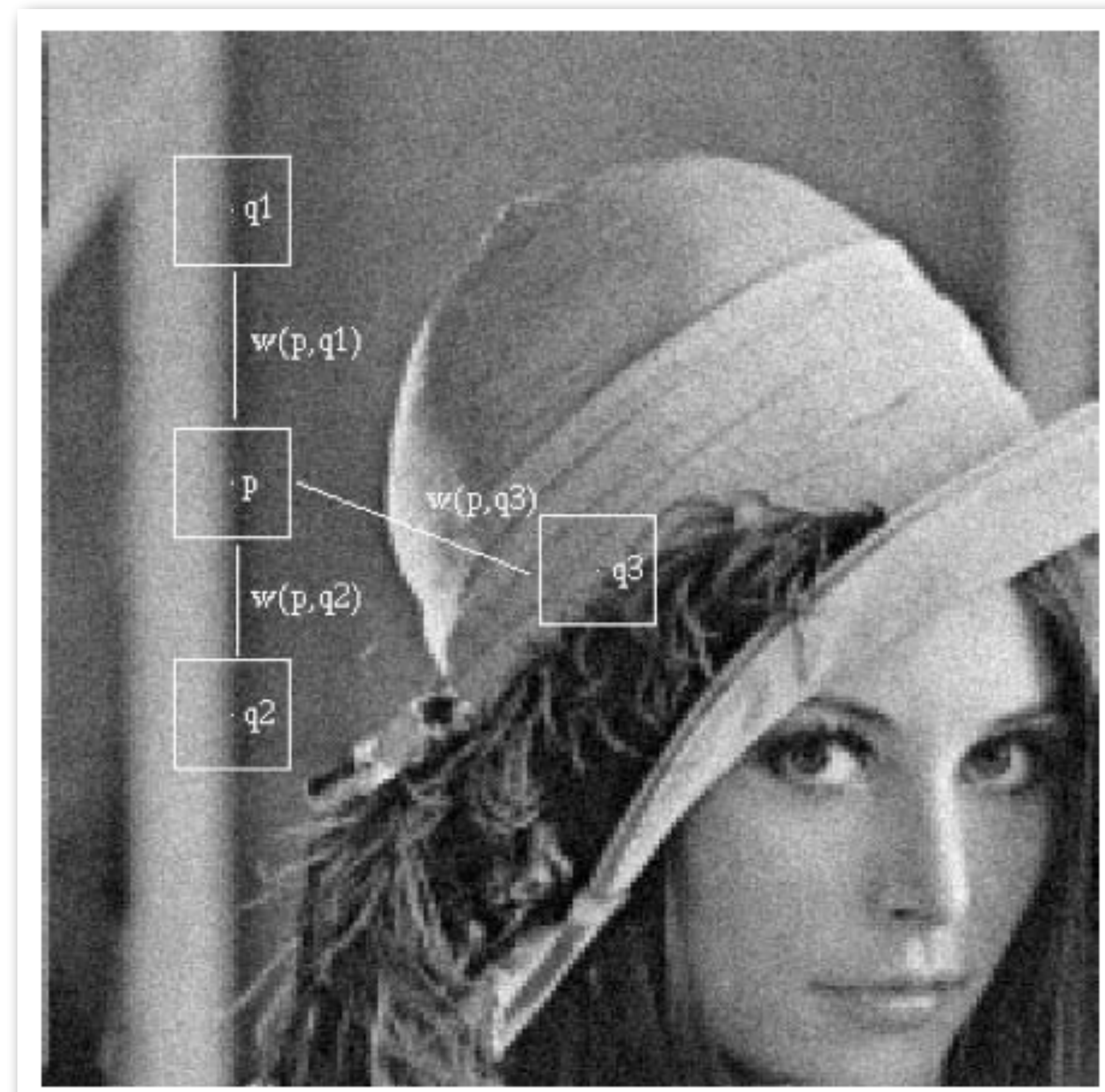
- N_p and N_q are vectors of pixel values in square window around pixels *p* and *q* (highlighted regions in figure)
- Difference between N_p and P_q = "similarity" of surrounding regions (here: L2 distance)
- C*p* is a normalization constant to ensure weights sum to one for pixel *p*.
- S is the search region (given by dotted red line in figure)

Denoising using non-local means

- **Large weight for input pixels that have similar neighborhood as *p***
  - Intuition: "filtered result is the average of pixels like this one"
  - In example below-right: *q1* and *q2* have high weight, *q3* has low weight



In each image pair above:
- Image at left shows the pixel to denoise.
- Image at right shows weights of pixels in 21x21-pixel kernel support window.



Buades et al. CVPR 2005

End of aside on image processing basics
(back to our simple camera pipeline)

Low light conditions need long exposure…
blur due to camera shake

Image credit: https://www.colorexpertsbd.com/blog/how-to-fix-blurry-photos-induced-by-camera-shake-in-photoshop

Low light photo: most regions underexposed (short exposure) to avoid blur + some region close to overexposed

Brightened image to see detail in dark regions, notice noise in dark regions

Attempt to denoise… splotchy effect remains

Walking people are blurred…

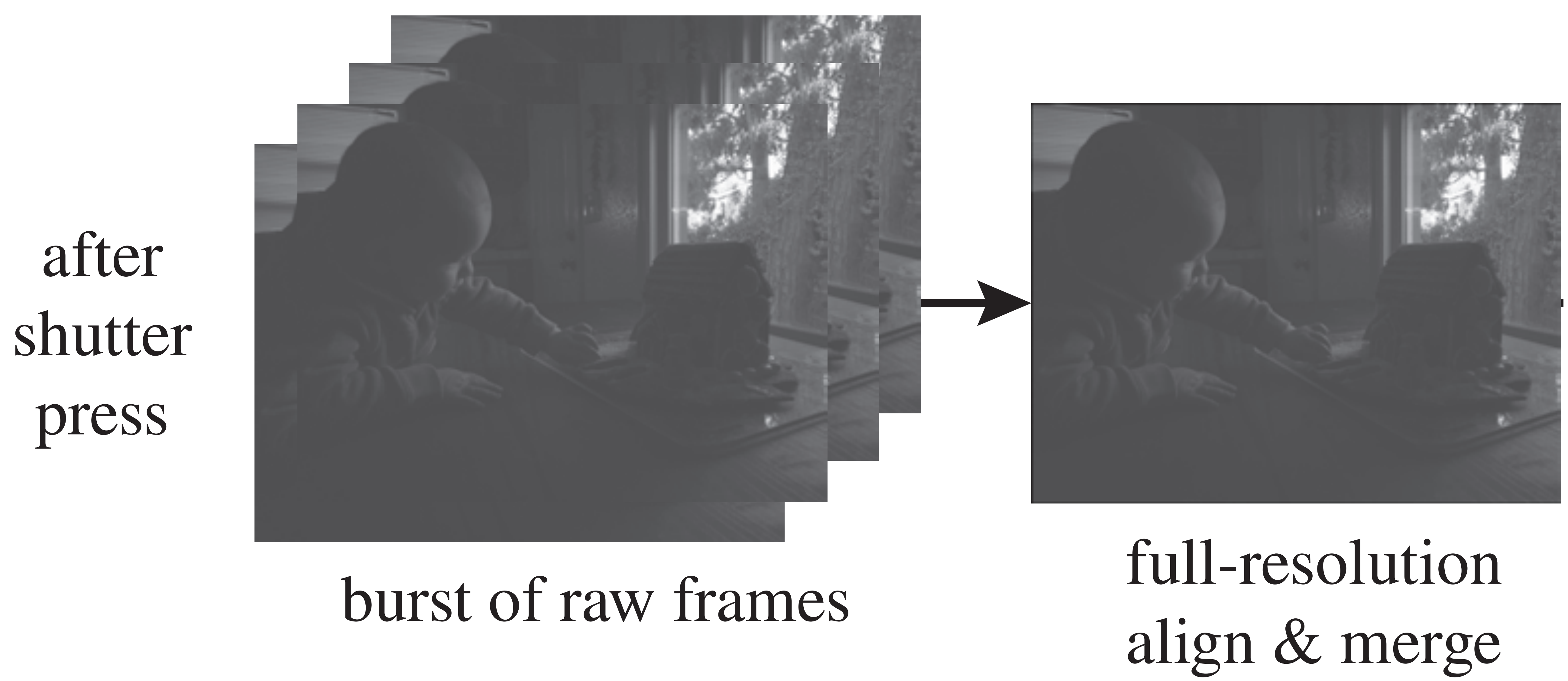
Walking people are blurred…

Also still significant noise.

Idea: merge sequence of captures

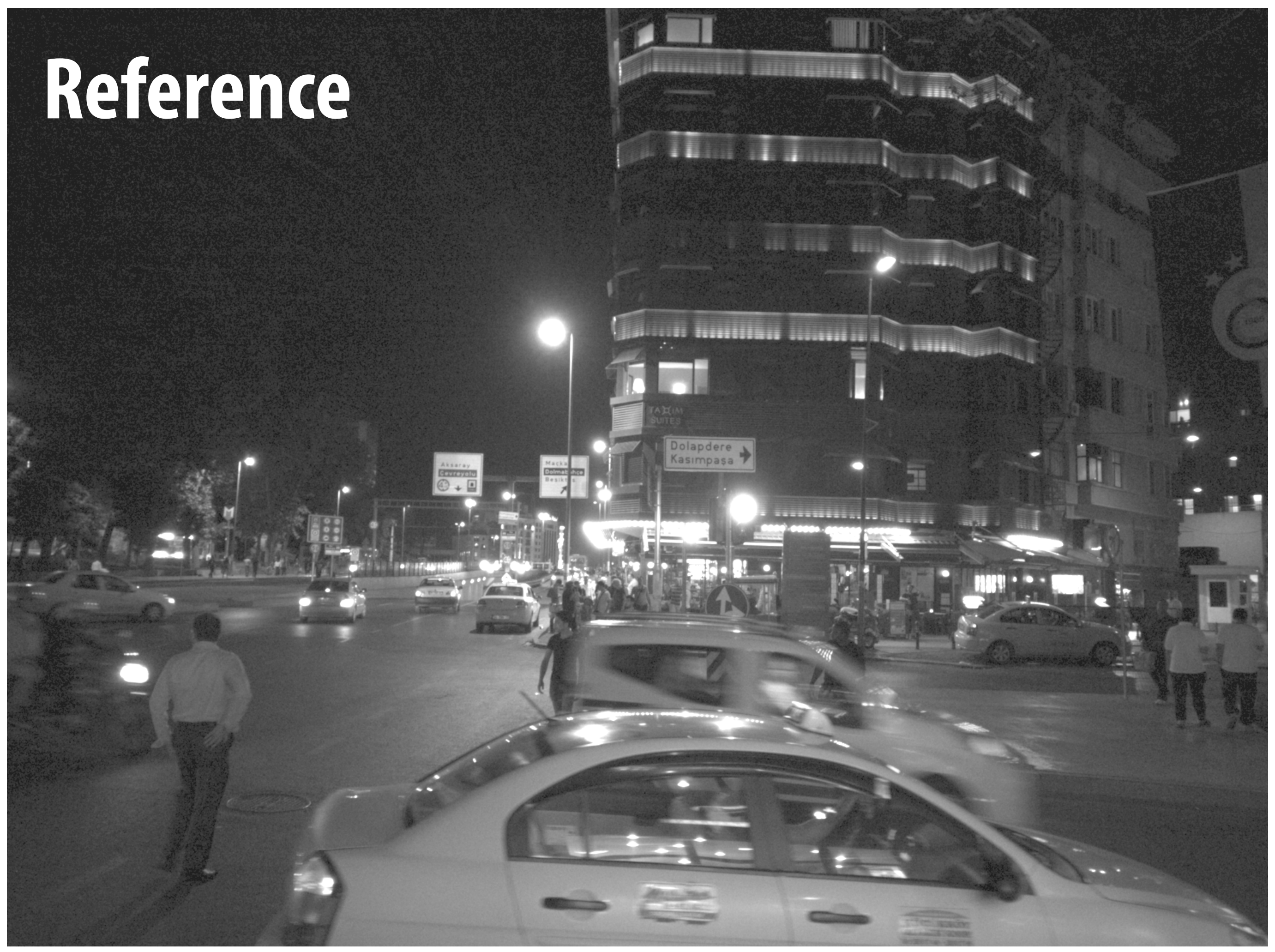
Algorithm used in Google Pixel Phones [Hasinoff 16]

- Long exposure: reduces noise (acquires more light), but introduces blur (camera shake or scene movement)
- Short exposure: sharper image, but lower signal/noise ratio
- Idea: take sequence of shorter exposures, but align images in software, then merge them into a single sharp image with high signal to noise ratio

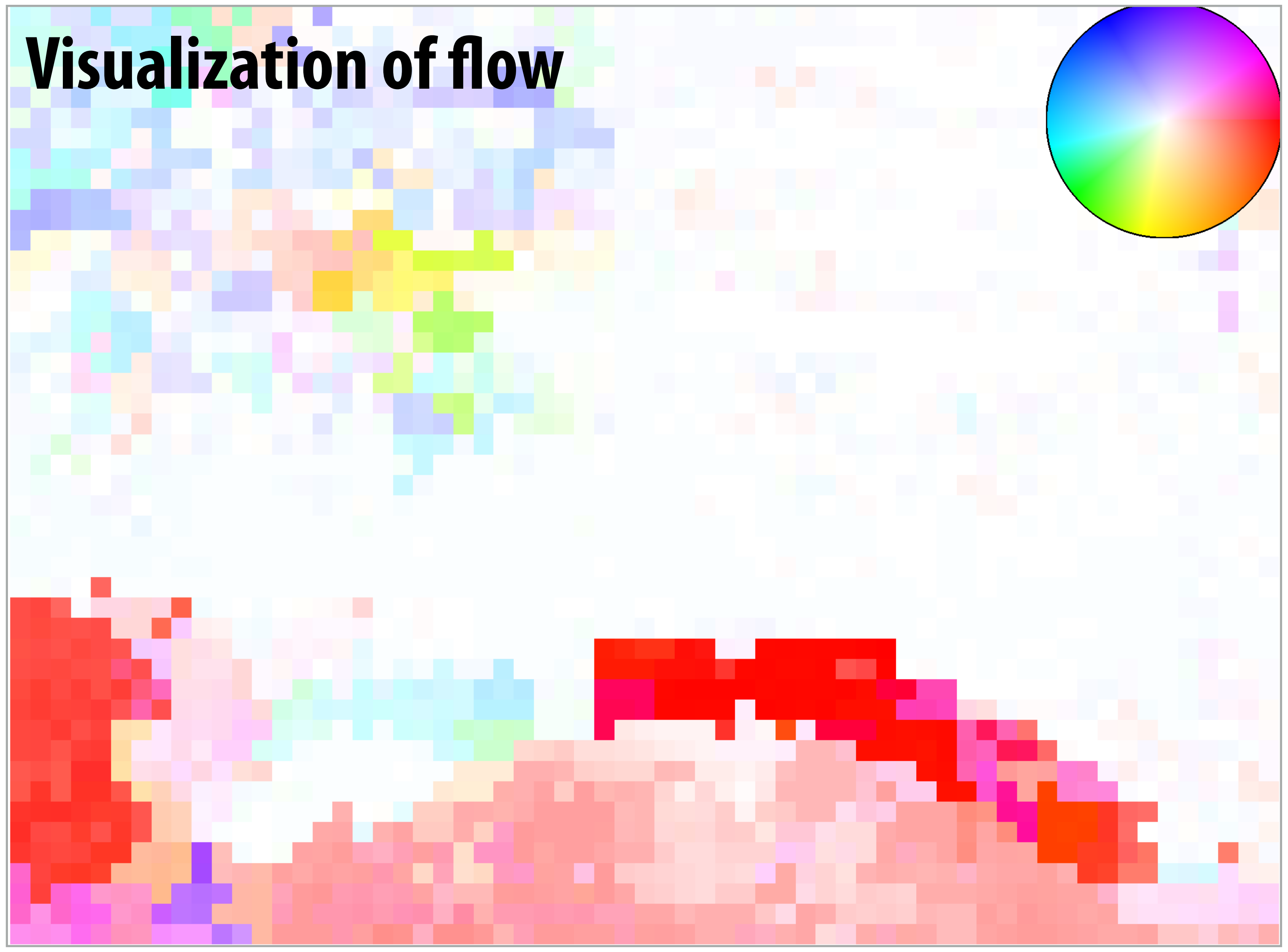


Align and merge algorithm

Image pair



- For each image in burst, align to reference frame (use sharpest photo as reference frame)
  - Compute optical flow field aligning image pair
- Simple merge algorithm: warp images according to flow, and sum
- More sophisticated techniques only merge pixels where confidence in alignment is (use noisy reference pixels when alignment fails)



[Image credit: Hasinoff 16]

Results of align and merge

[Hasinoff 16]

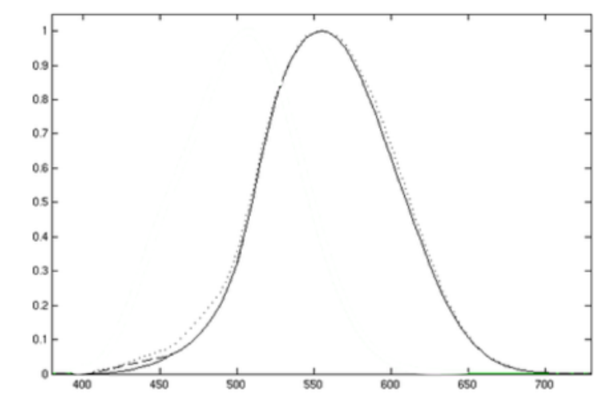
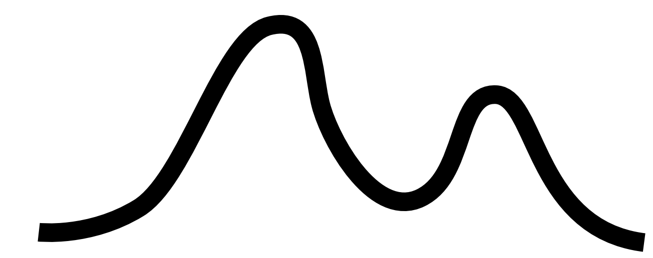
Details of alignment and merging algorithm in tonight's reading (and assignment 1)

(a) *Reference frame* **(b)** *Temporal mean* **(c)** *Temporal mean with alignment* **(d)** *Robust merge with alignment*

[Image credit: Hasinoff 16]

Gamma correction
(global tone adjustment)

Lightness (perceived brightness) aka luma

Lightness (L*) $\xleftarrow{?}$ **Luminance (Y)** $= \int_\lambda$  $*$ 

(Perceived by brain)

(Response of eye)

Spectral sensitivity of eye (eye's response curve)

Radiance (energy spectrum from scene)

Dark adapted eye: $L^* \propto Y^{0.4}$

Bright adapted eye: $L^* \propto Y^{0.5}$

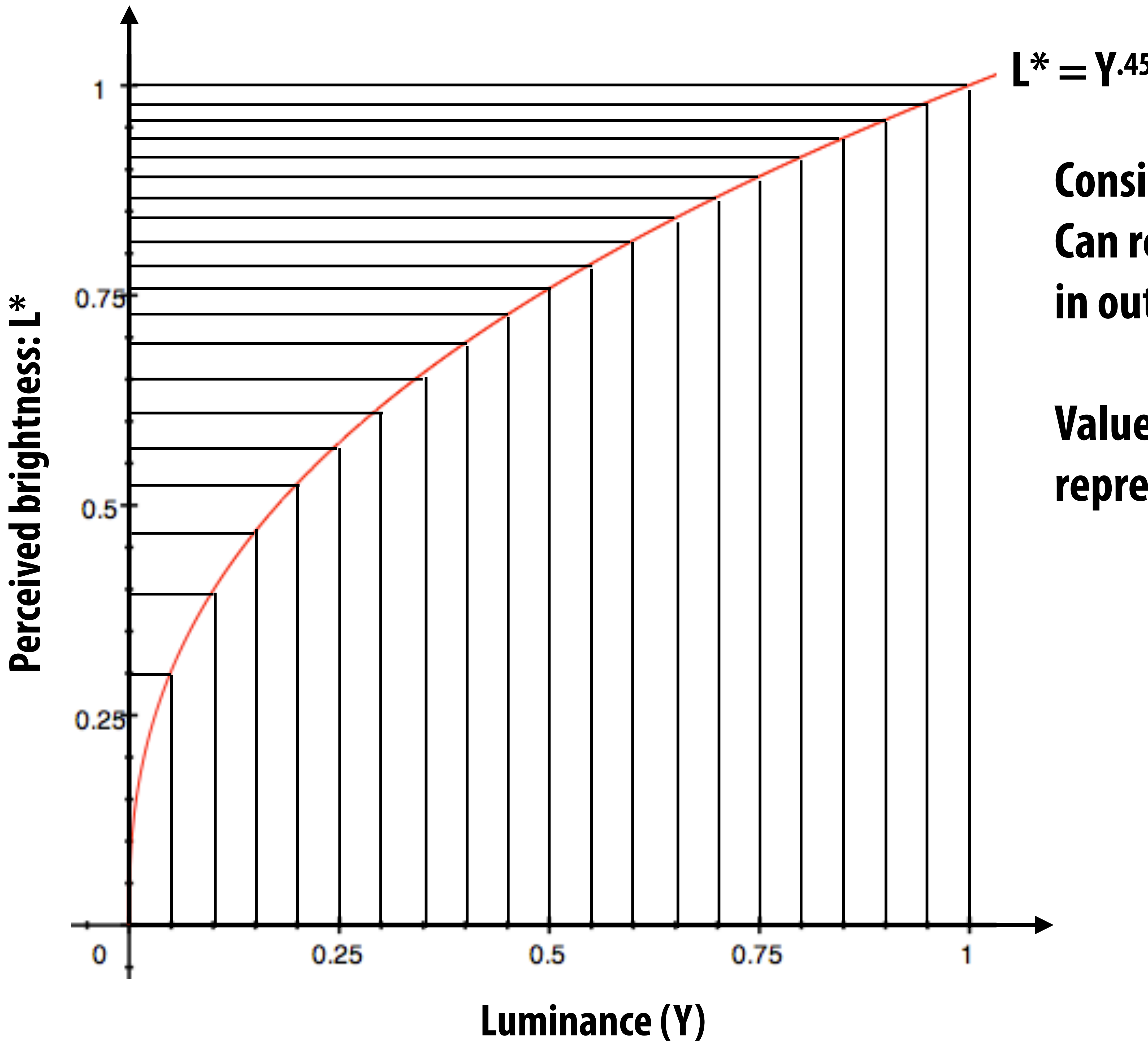
In a dark room, you turn on a light with luminance: Y_1

You turn on a second light that is identical to the first. Total output is now: $Y_2 = 2Y_1$

Total output appears $2^{0.4} = 1.319$ **times brighter to dark-adapted human**

Note: Lightness (L*) is often referred to as luma (Y')

Consider an image with pixel values encoding luminance (linear in energy hitting sensor)



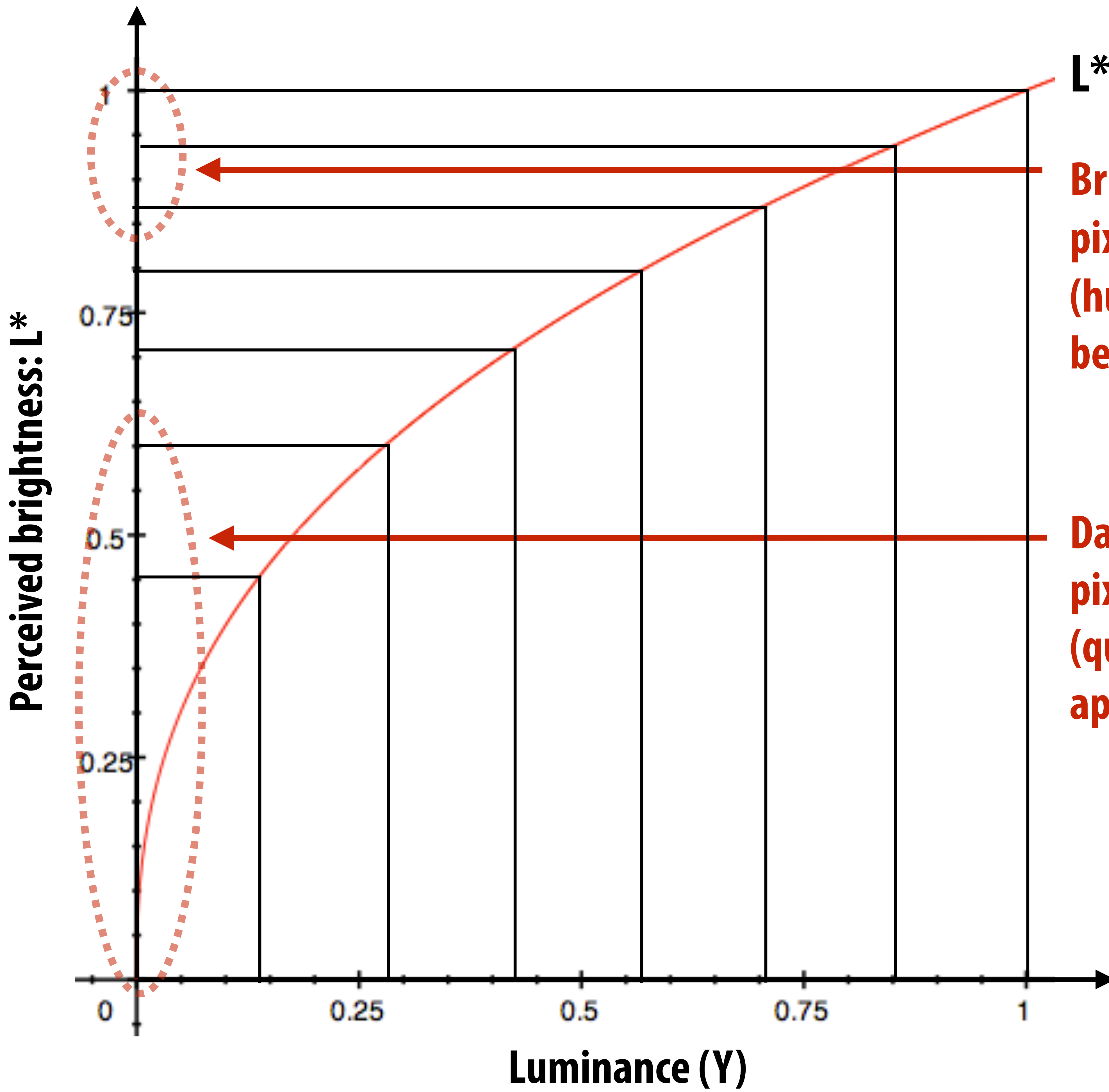
Consider 12-bit sensor pixel:
Can represent 4096 unique luminance values in output image

Values are ~ linear in luminance since they represent the sensor's response

Problem: quantization error

Many common image formats store 8 bits per channel (256 unique values)

Insufficient precision to represent brightness in darker regions of image



$L^* = Y^{.45}$

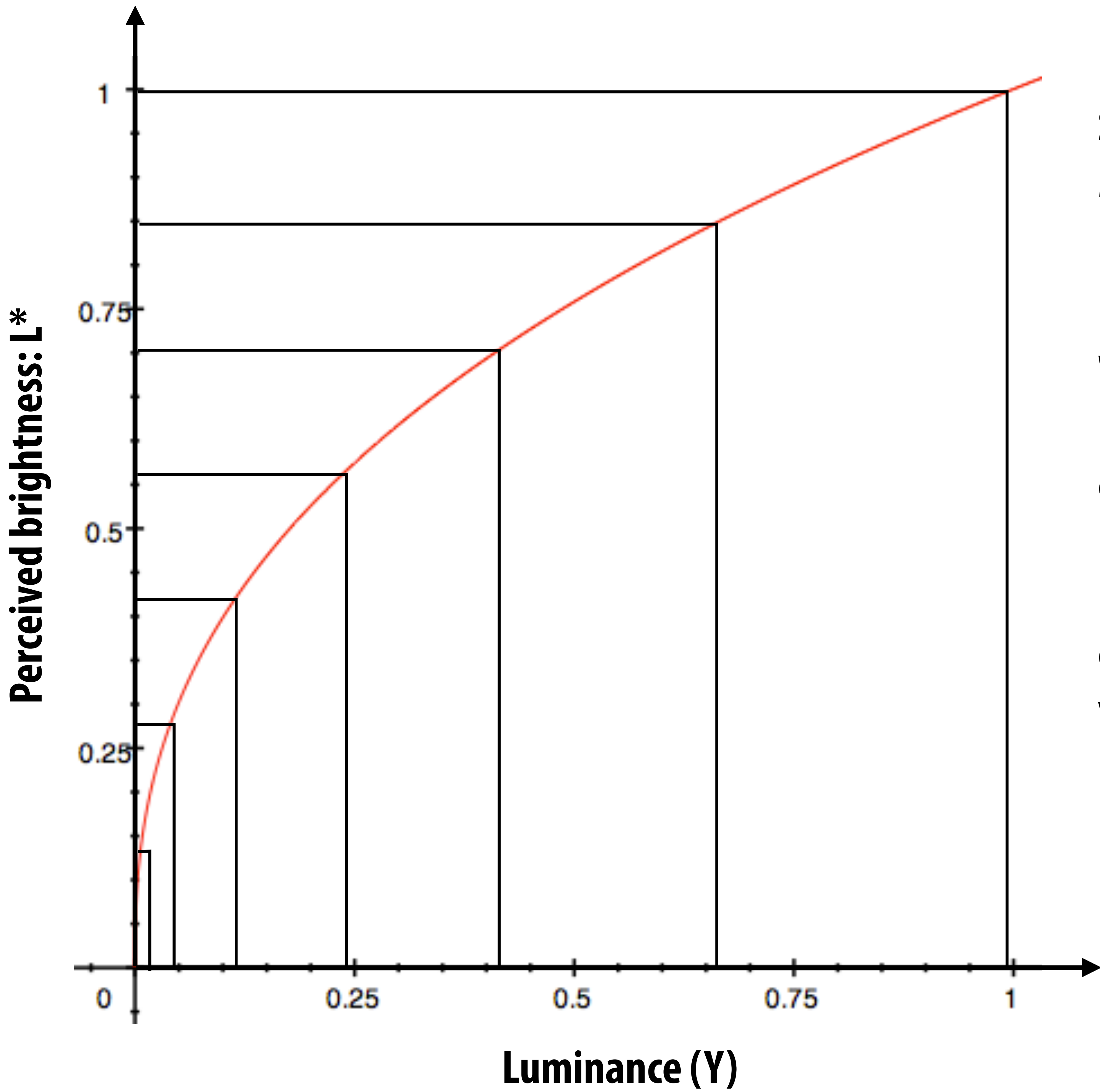
Bright regions of image: perceived difference between pixels that differ by one step in luminance is small! (human may not even be able to perceive difference between pixels that differ by one step in luminance!)

Dark regions of image: perceived difference between pixels that differ by one step in luminance is large! (quantization error: gradients in luminance will not appear smooth.)

Rule of thumb: human eye cannot differentiate <1% differences in luminance

Store lightness in 8-bit value, not luminance

Idea: distribute representable pixel values evenly with respect to perceived brightness, not evenly in luminance (make more efficient use of available bits)



Solution: pixel stores $Y^{0.45}$
Must compute $(\text{pixel\_value})^{2.2}$ prior to display on LCD

Warning: must take caution with subsequent pixel processing operations once pixels are encoded in a space that is not linear in luminance.

e.g., When adding images should you add pixel values that are encoded as lightness or as luminance?

Local-tone adjustment

Weights

Improve picture's aesthetics by locally adjusting contrast, boosting dark regions, decreasing bright regions

(more details in the next lecture)

Combined image
(unique weights per pixel)

Image credit: Mertens 2007

Summary: simplified image processing pipeline

- Correct pixel defects
- Align and merge
- Correct for sensor bias (using measurements of optically black pixels)
- Vignetting compensation
- White balance

(10-12 bits per pixel)
1 intensity value per pixel
Pixel values linear in energy

- Demosaic
- Denoise
- Gamma Correction (non-linear mapping)

3x12 bits per pixel
RGB intensity per pixel
Pixel values linear in energy

- Local tone mapping
- Final adjustments sharpen, fix chromatic aberrations, hue adjust, etc.

3x8-bits per pixel
Pixel values perceptually linear