Lecture 2:

The Camera Image Processing Pipeline

Visual Computing Systems

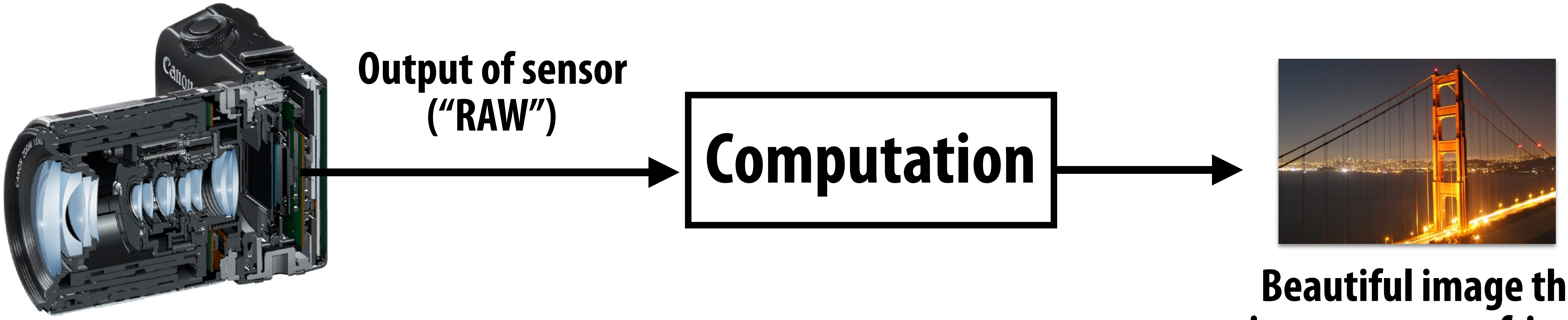
Stanford CS348V, Winter 2018

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

Today's lecture

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

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



Output of sensor ("RAW")

Computation

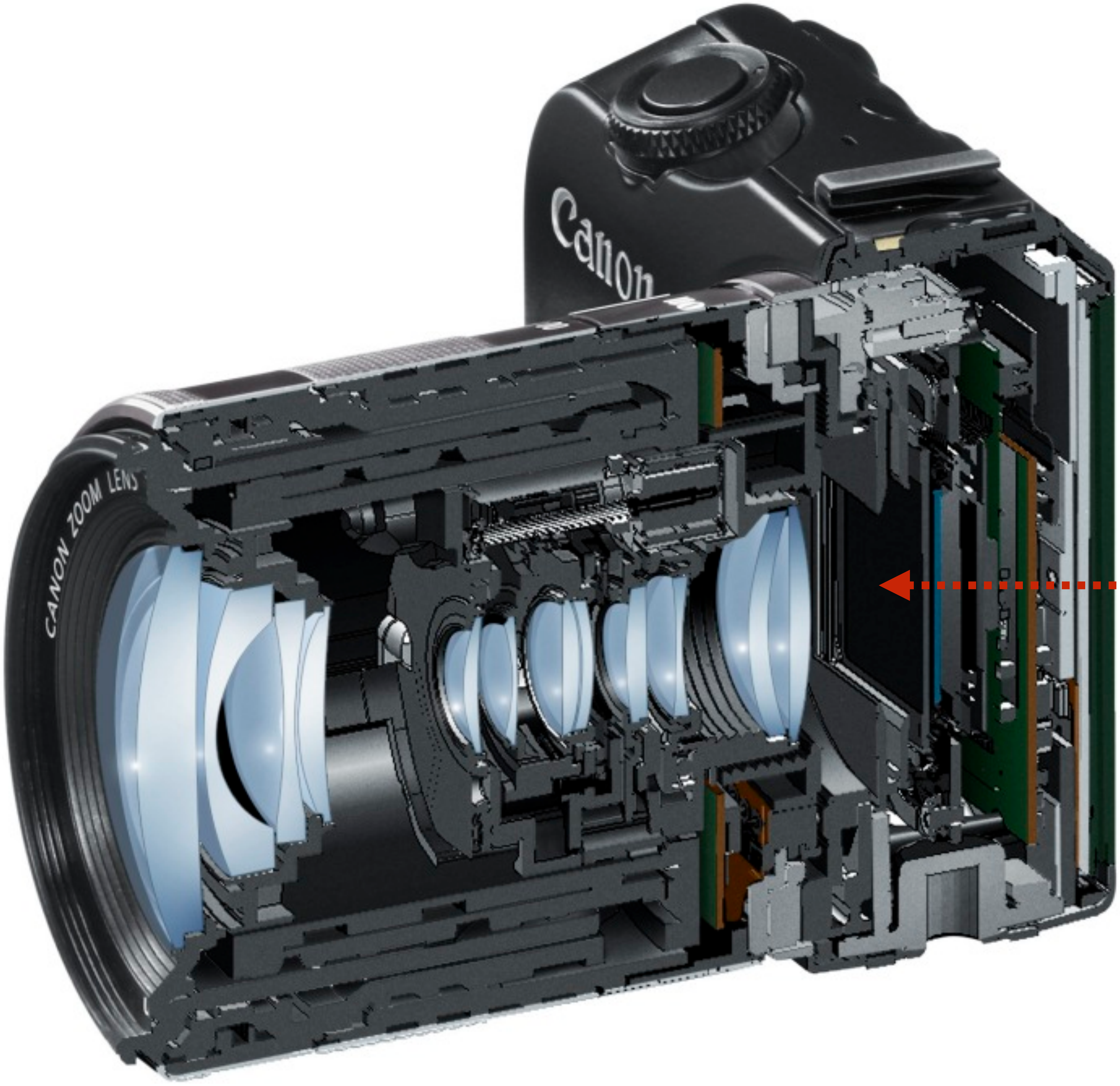
Beautiful image that impresses your friends on Instagram

Where we are headed

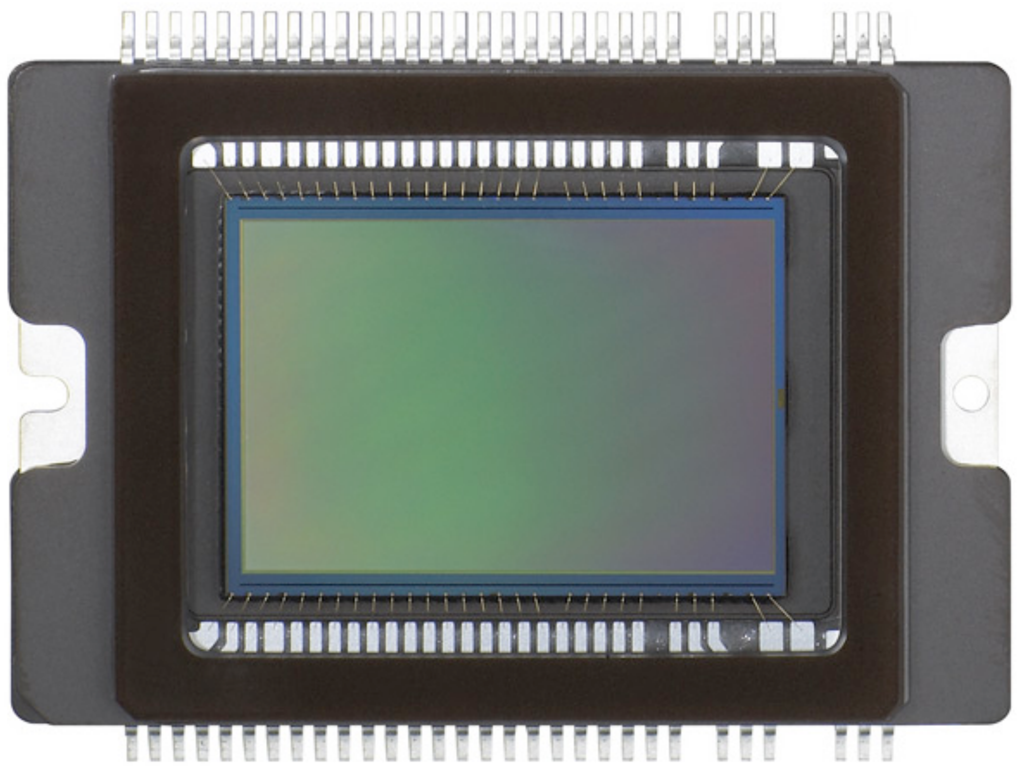
- **I'm about to describe a sequence of operations that take raw image pixels from a sensor (measurements of light) to high-quality pictures**
  - Correct for sensor bias (using measurements of optically black pixels)
  - Correct pixel defects
  - Vignetting compensation
  - Dark frame subtract (optional)
  - White balance
  - Demosaic
  - Denoise / sharpen, etc.
  - Color Space Conversion
  - Gamma Correction
  - Color Space Conversion (Y'CbCr)
  - …

- **Today's pipelines are sophisticated, but they only scratch the surface of what future image processing pipelines might do**
  - Person identification, action recognition, scene understanding (to automatically compose shot or automatically pick best picture) etc.

Camera cross section



Sensor

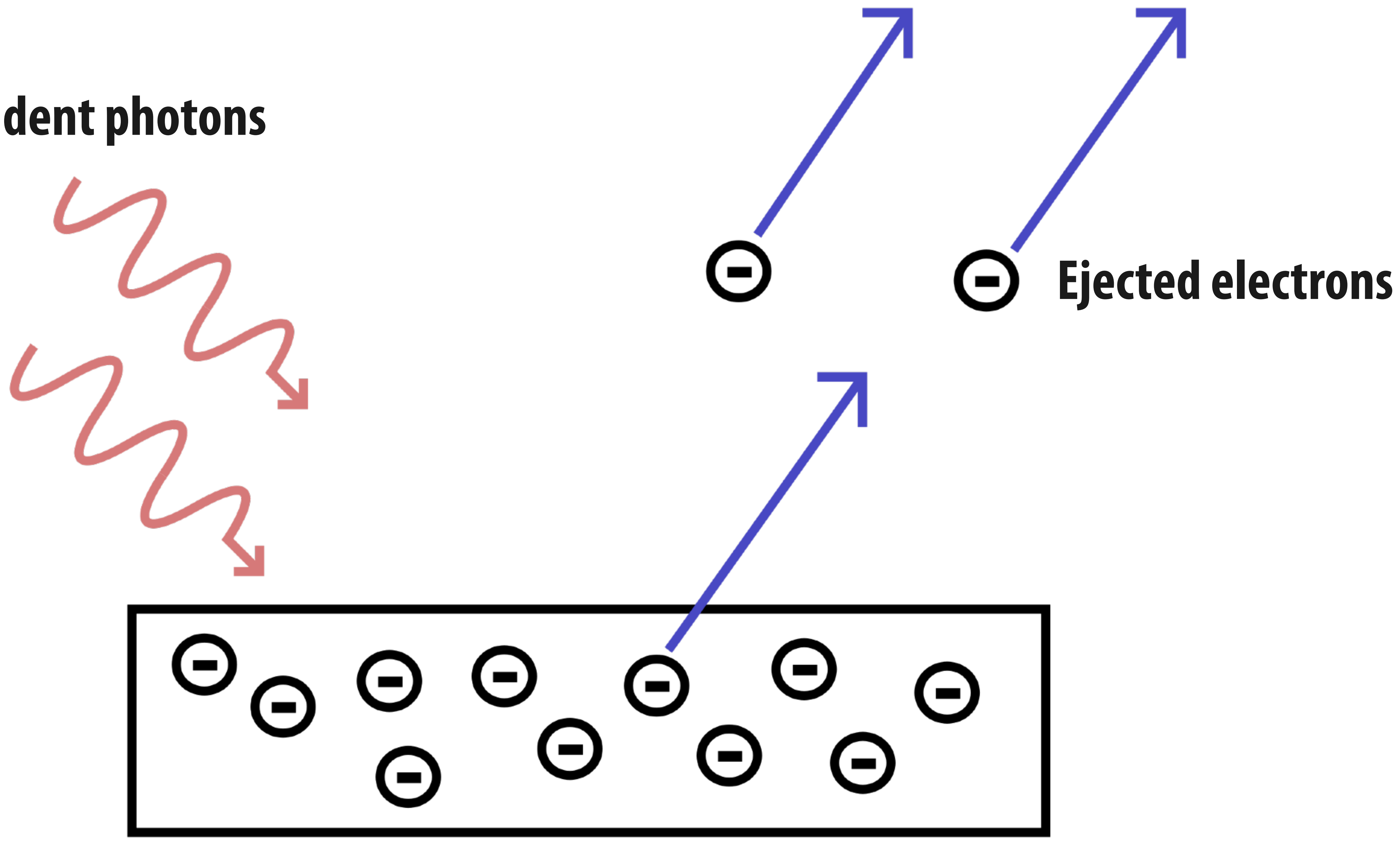


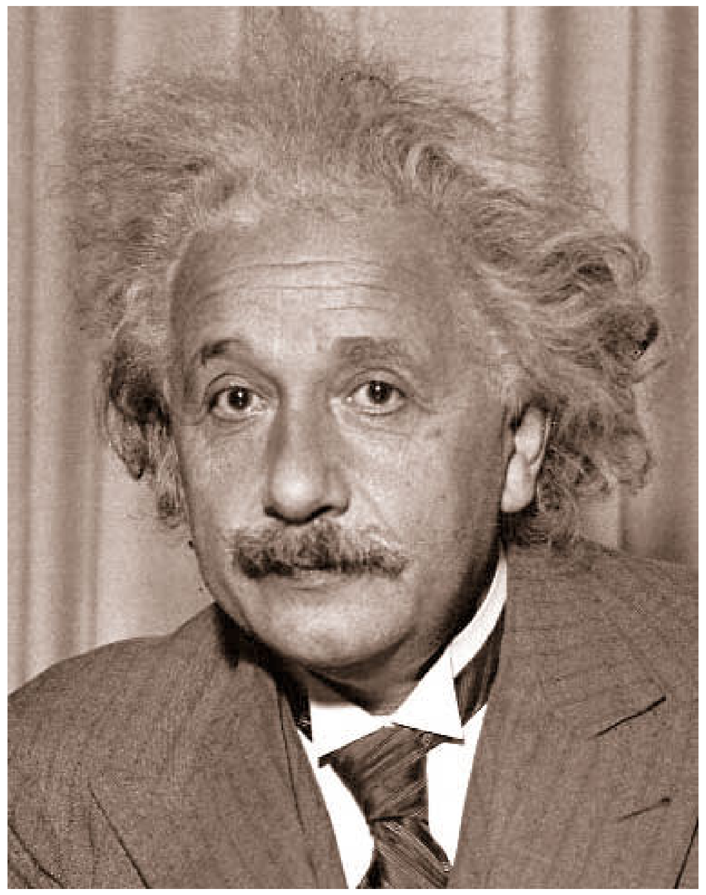
Canon 14 MP CMOS Sensor
(14 bits per pixel)

Image credit: Canon (EOS M)

The Sensor

Photoelectric effect

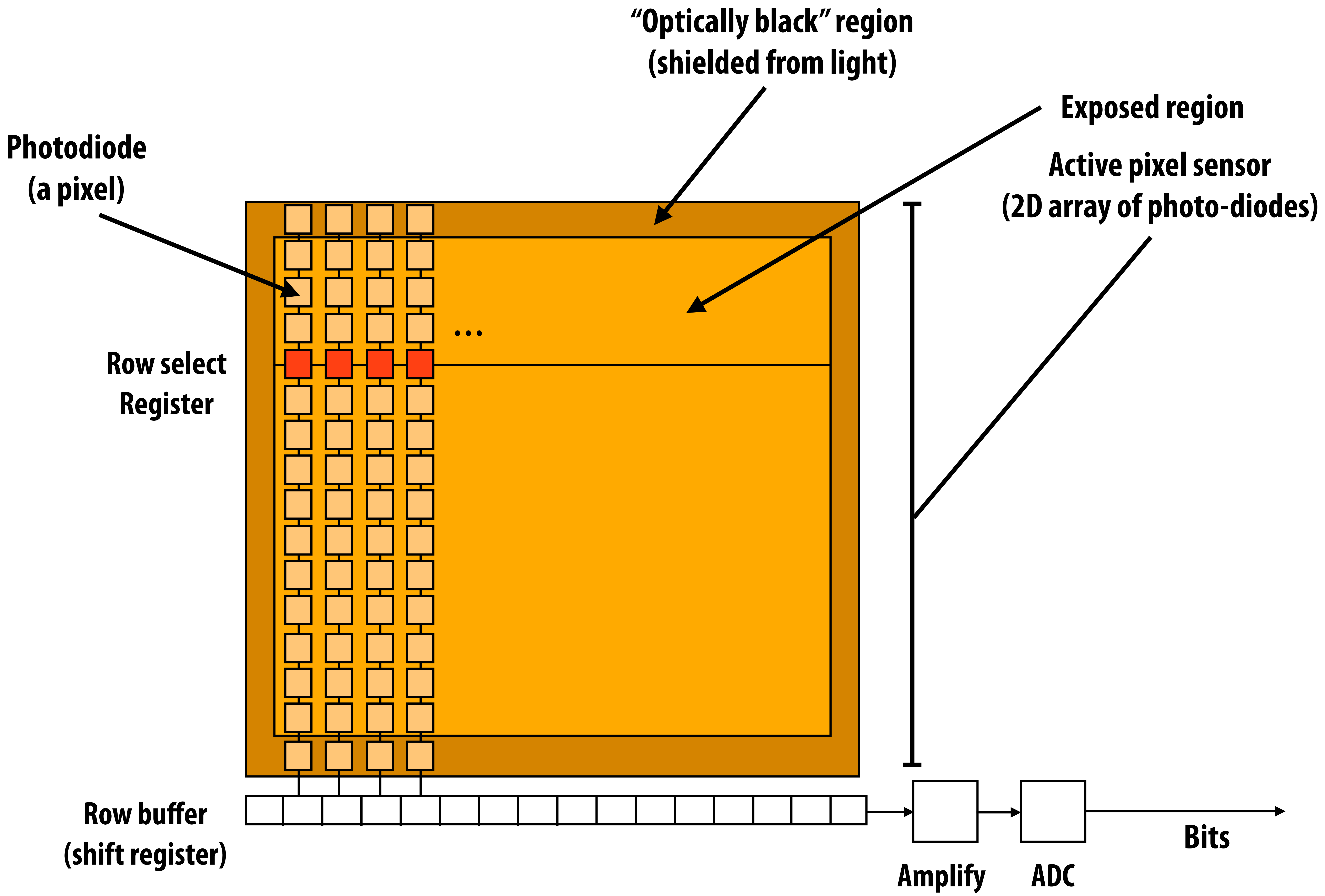



Albert Einstein

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
Photodiode
(a pixel)
Active pixel sensor
(2D array of photo-diodes)
...
Row select
Register
Row buffer
(shift register)
Amplify
ADC
Bits

CMOS APS (active pixel sensor) pixel

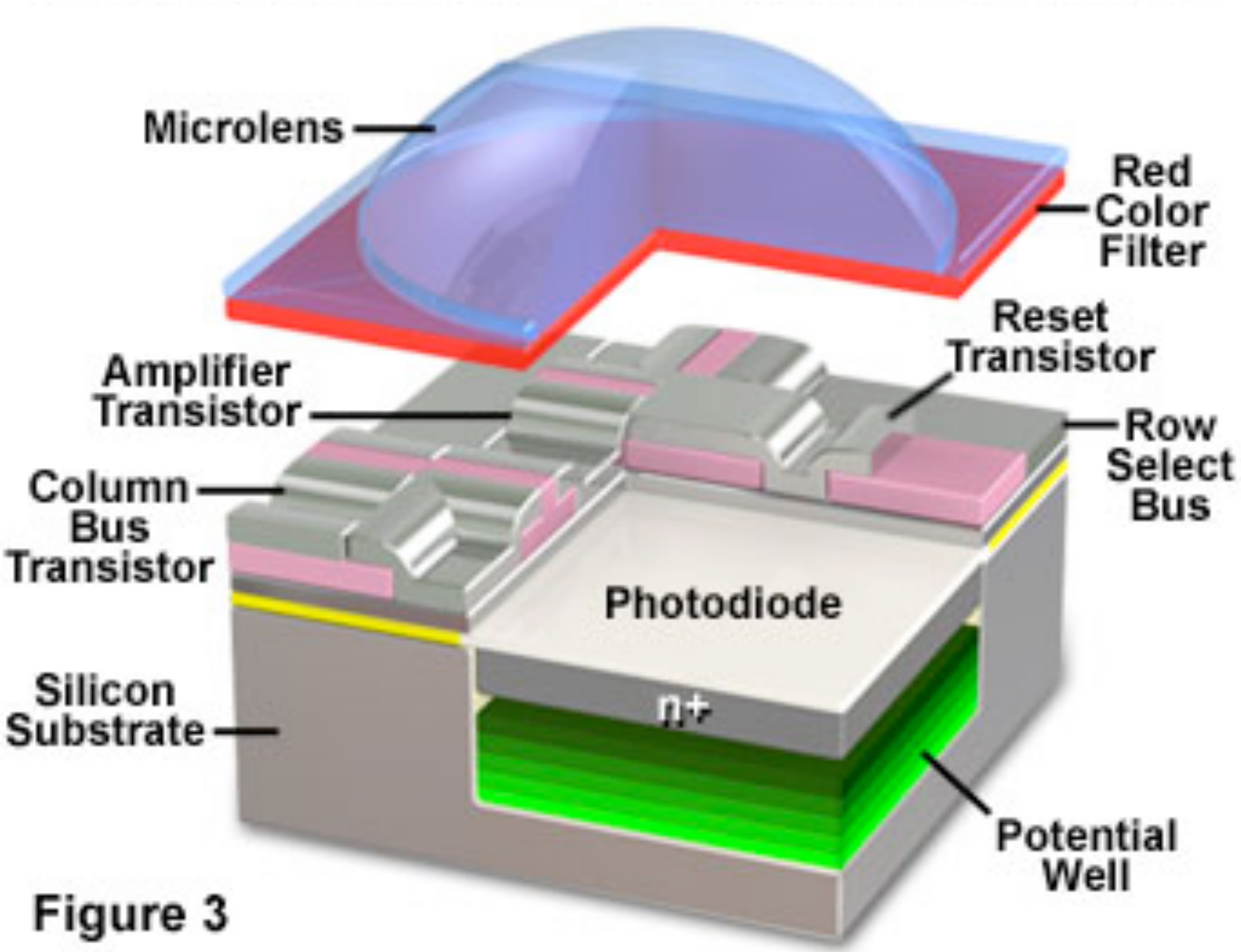
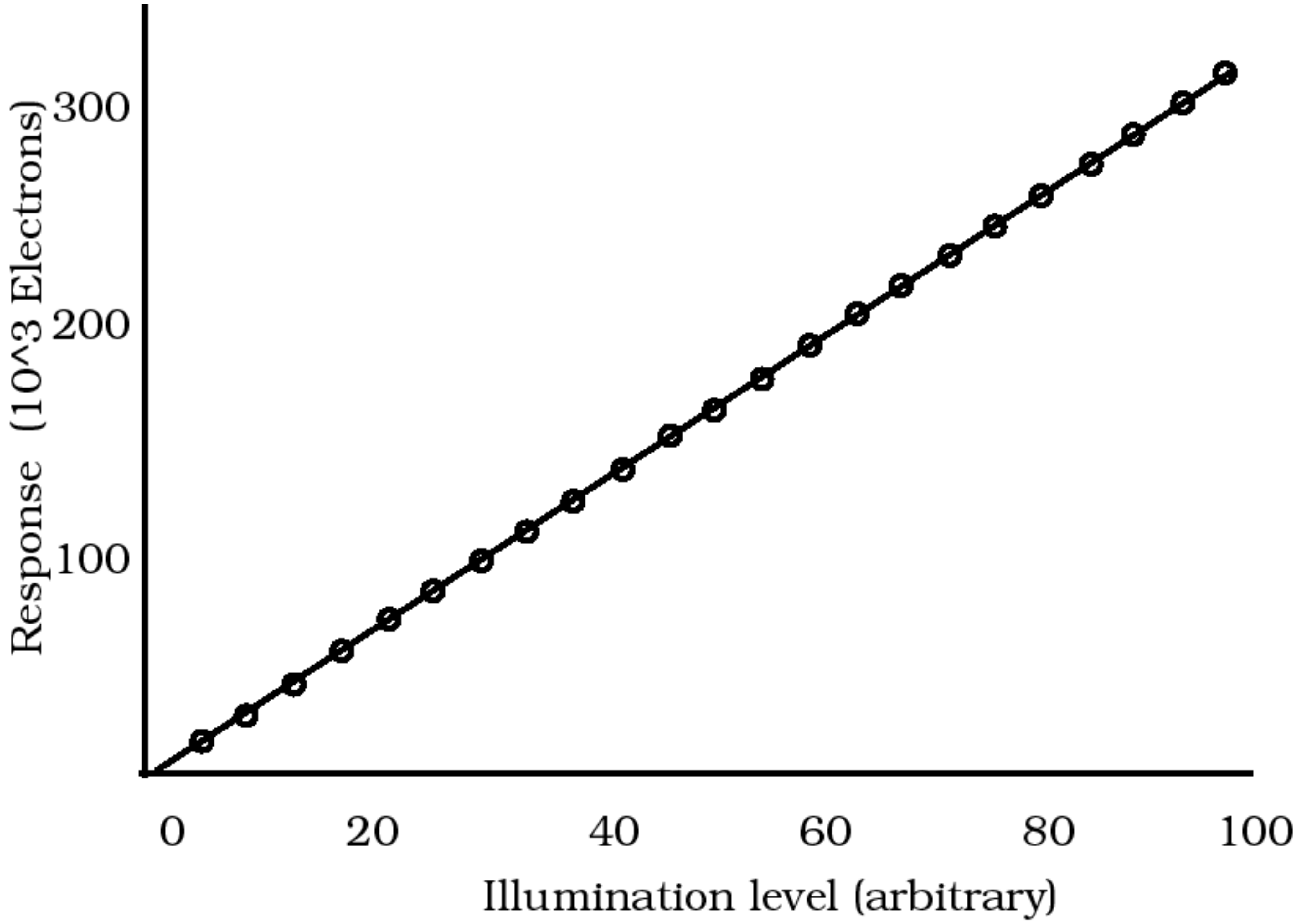


Illustration credit: Molecular Expressions (http://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html)

CMOS response functions are linear

Photoelectric effect in silicon:

- **Response function from photons to electrons is linear**
- **May have 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:

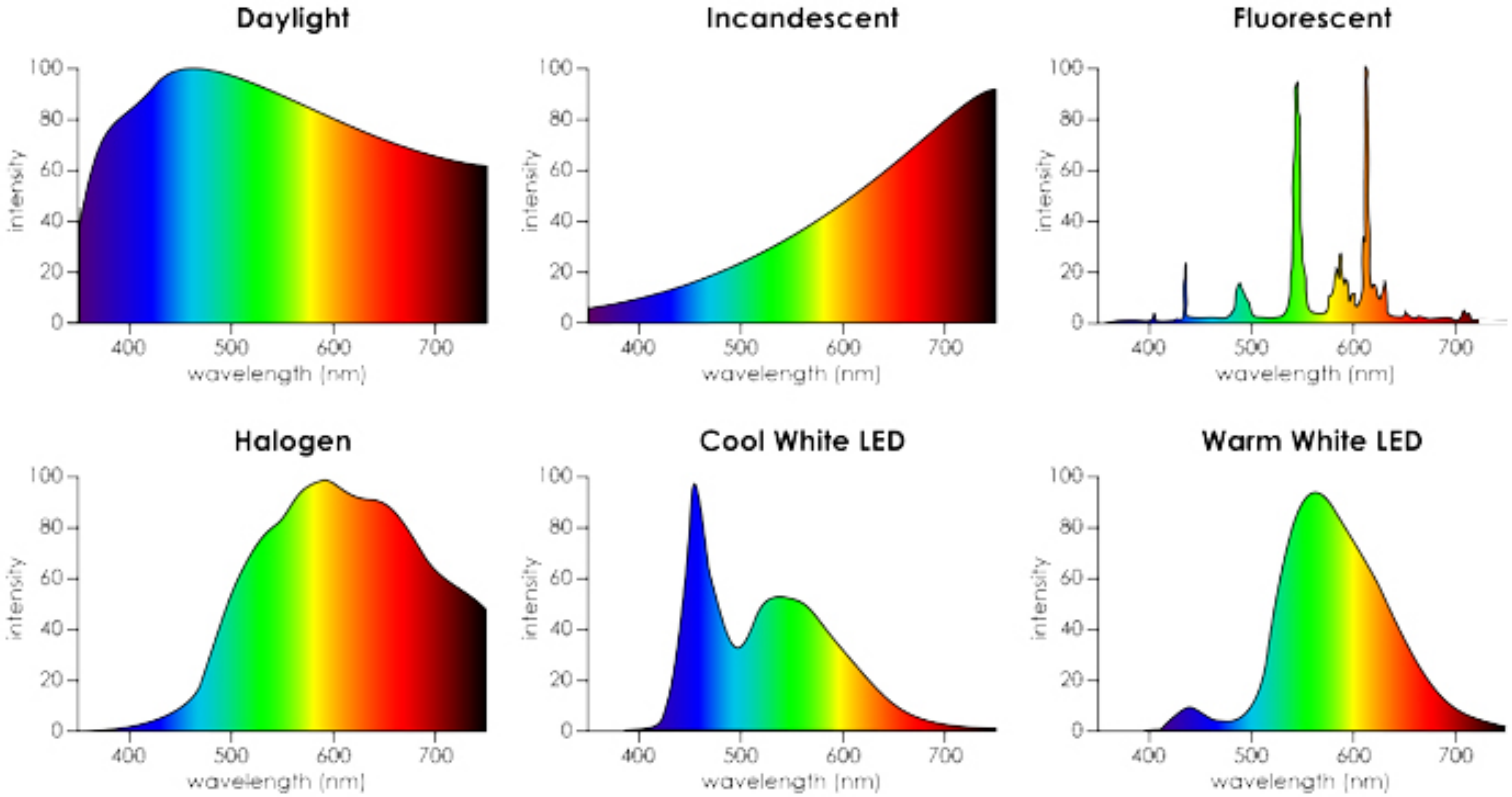


Figure credit:

admesy
ADVANCED MEASUREMENT SYSTEMS

Warm white vs. cool white

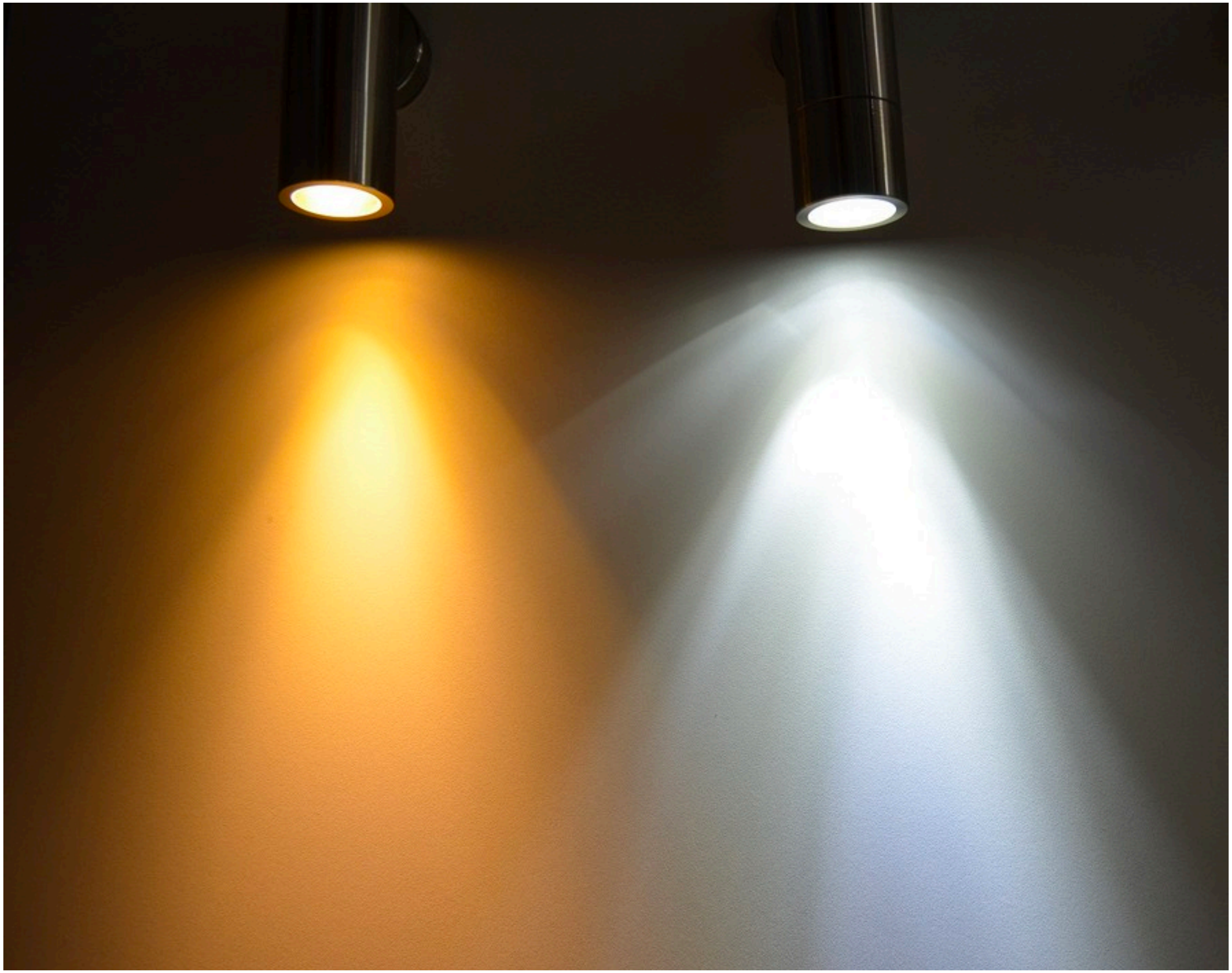


Image credit: (Oz Lighting: https://www.ozlighting.com.au/blog/what-is-warm-white-versus-cool-white/)

Photosensor response

- **Photosensor input: light**
  - Electromagnetic power distribution over wavelengths: $\Phi(\lambda)$
- **Photosensor output: a “response” … a number**
  - e.g., encoded in electrical signal
- **Spectral response function: $f(\lambda)$**
  - Sensitivity of sensor to light of a given wavelength
  - Greater $f(\lambda)$ corresponds to more a efficient sensor (when $f(\lambda)$ is large, a small amount of light at wavelength λ will trigger a large sensor response)
- **Total response of photosensor:**

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

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)

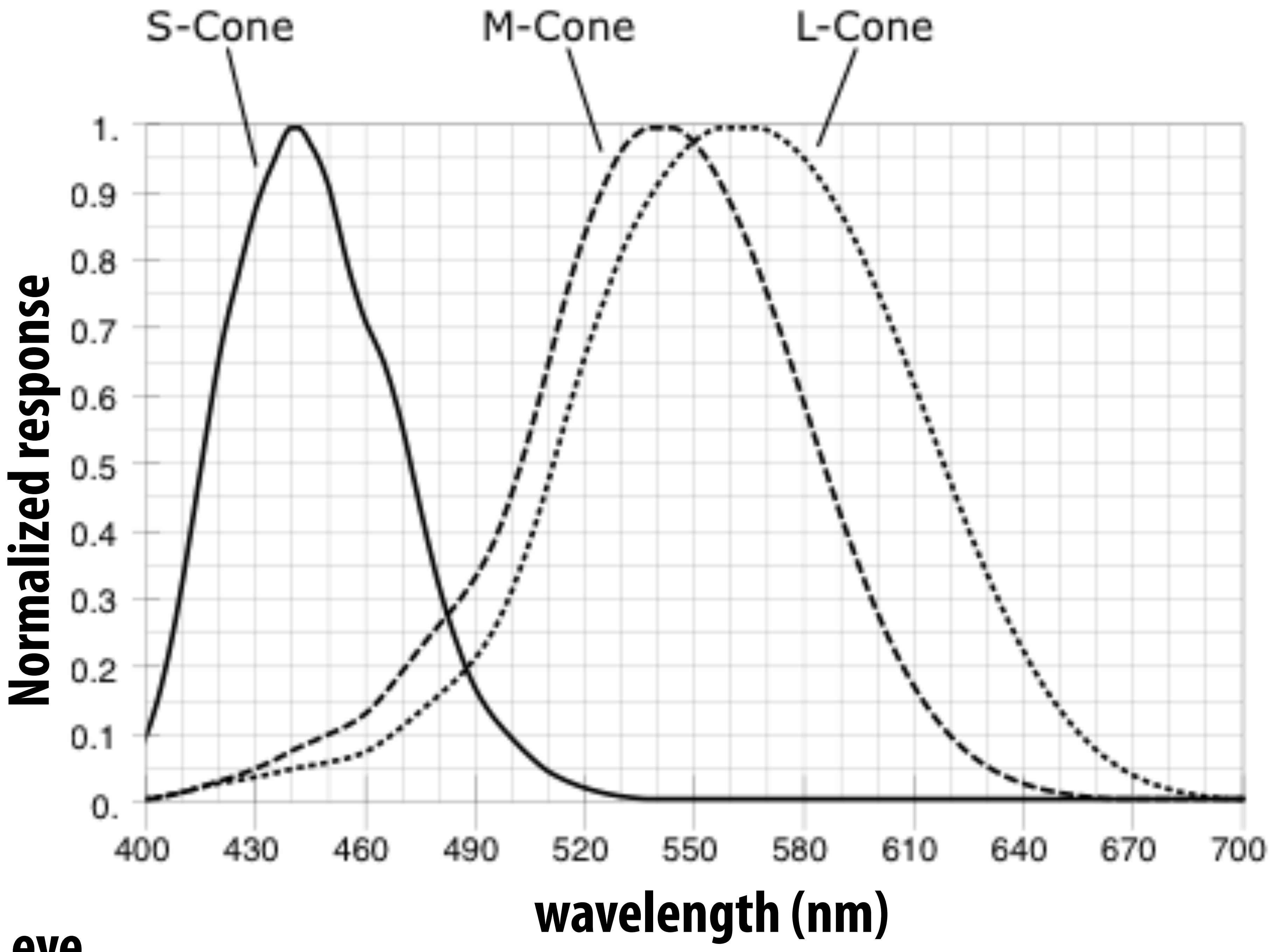
Implication: the space of human-perceivable colors is three dimensional

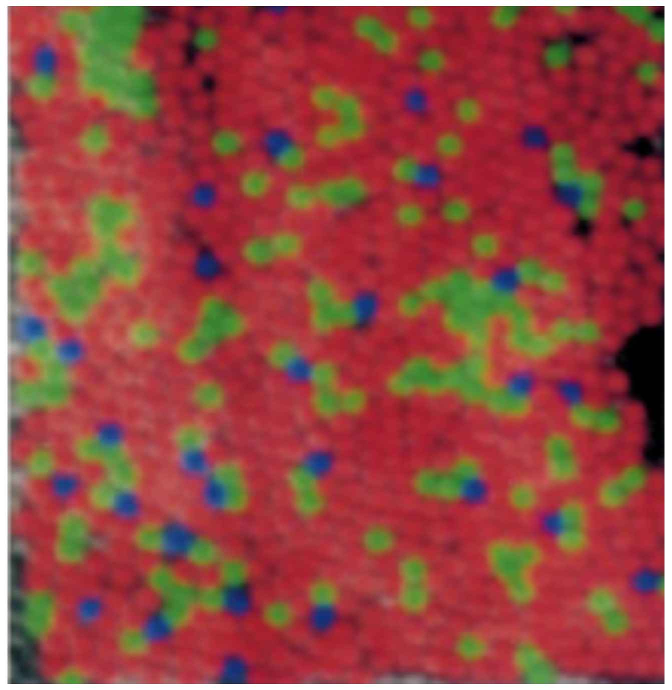
$$S = \int_\lambda \Phi(\lambda) S(\lambda) d\lambda$$

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

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

Response functions for S, M, and L cones

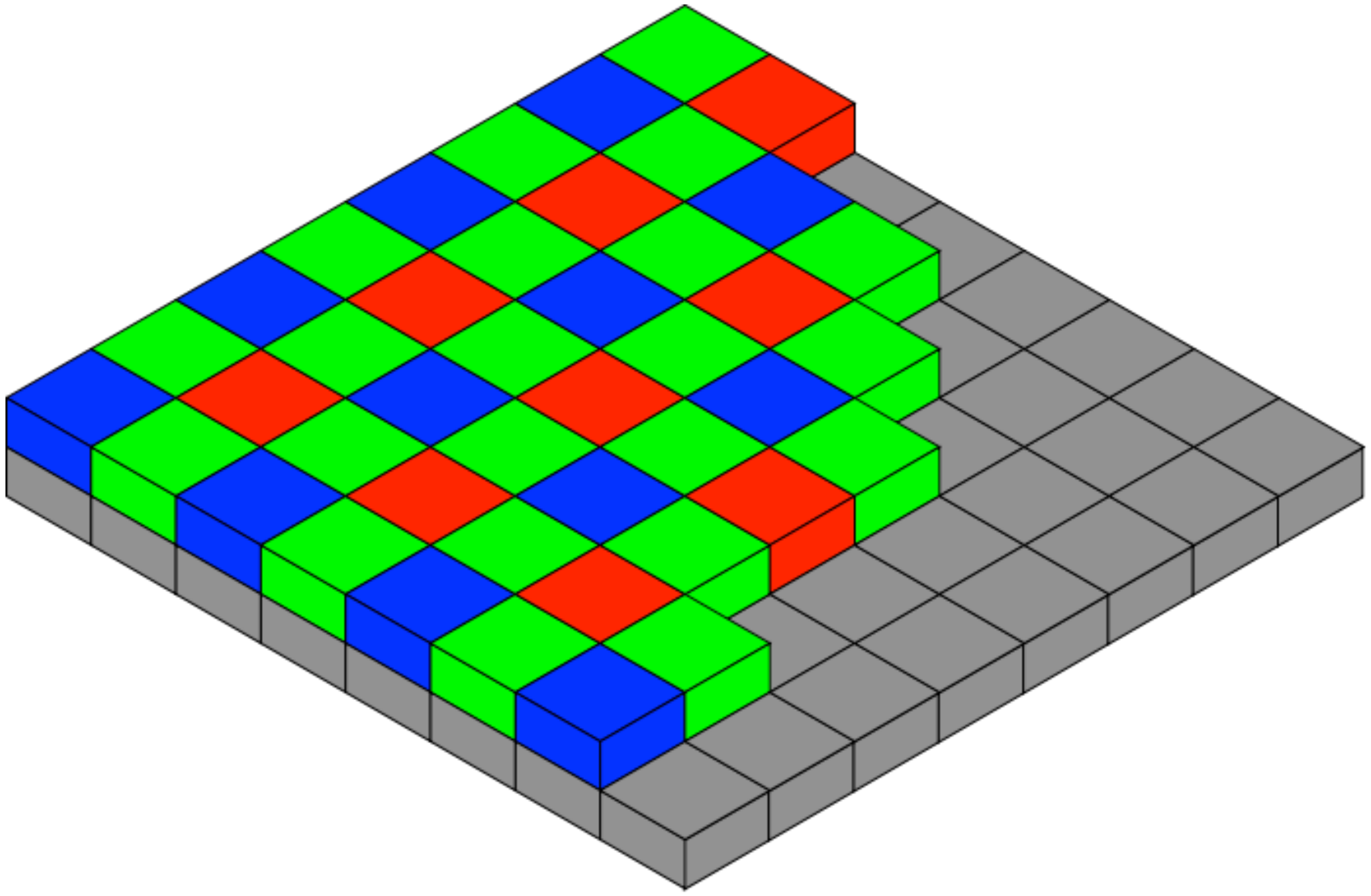




Uneven distribution of cone types in eye
~64% of cones are L cones, ~ 32% M cones

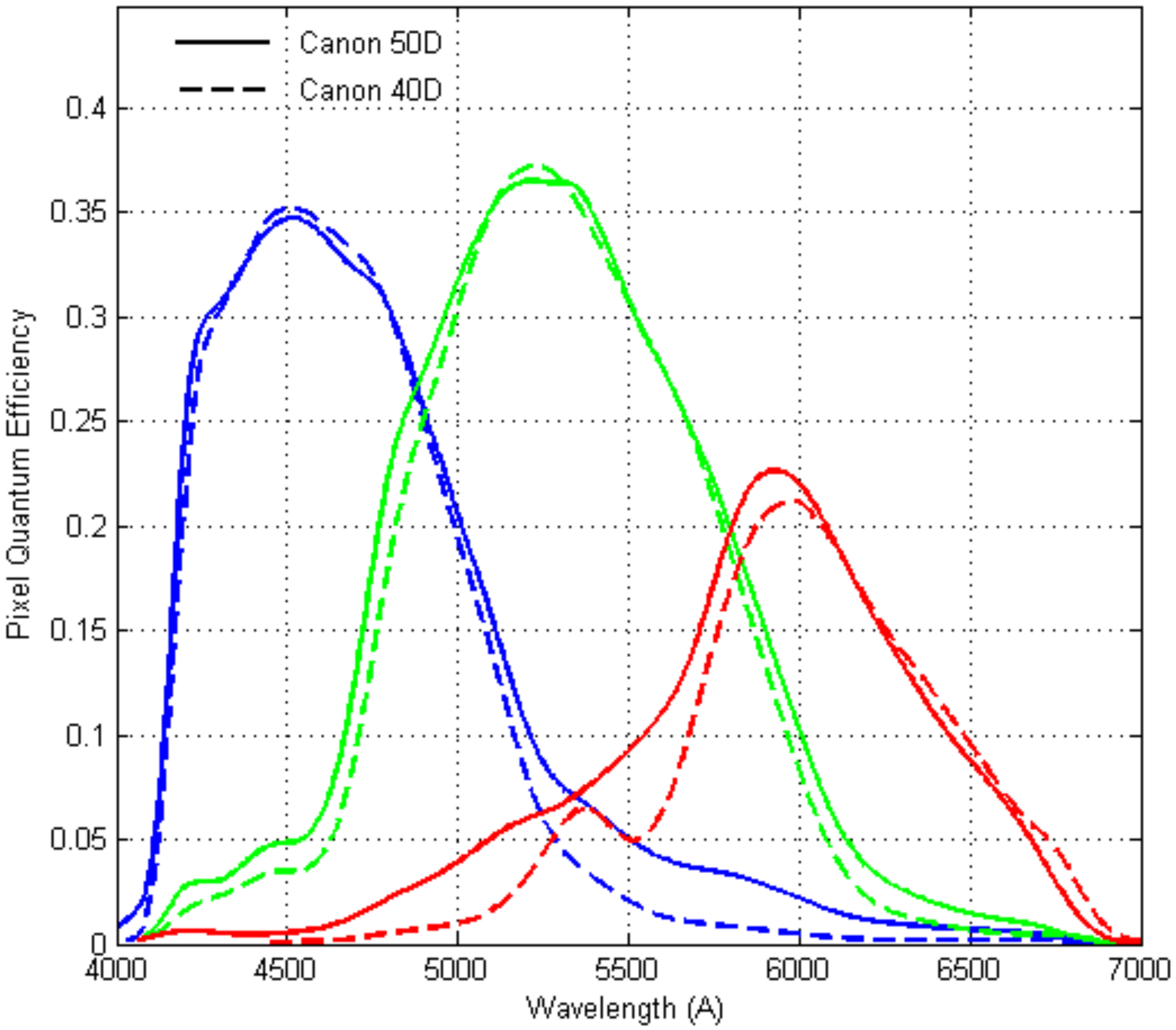
Color filter array (Bayer mosaic)

- 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



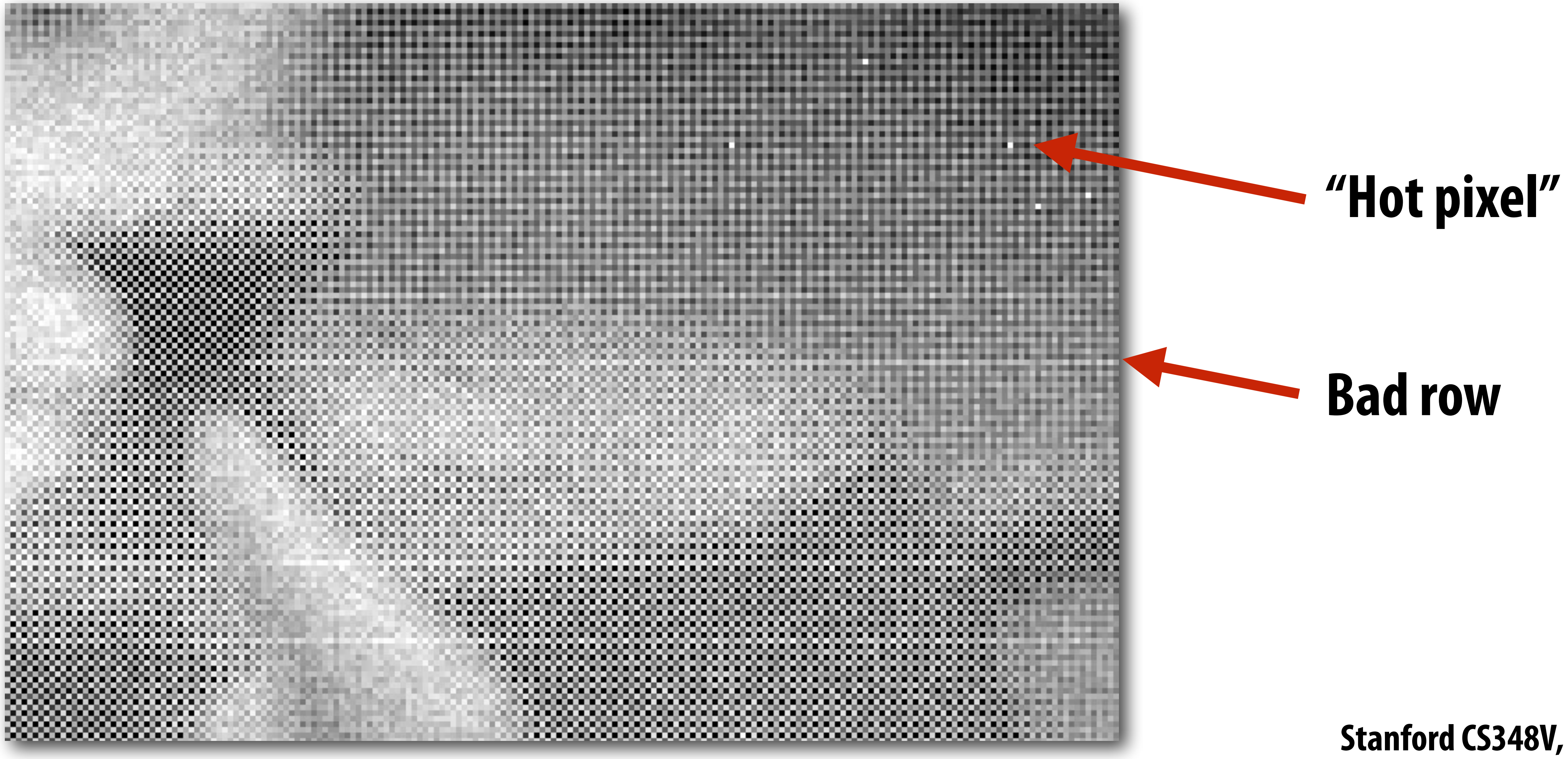
$$f(\lambda)$$

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

RAW sensor output (simulated data)

Light Hitting Sensor

RAW output of sensor



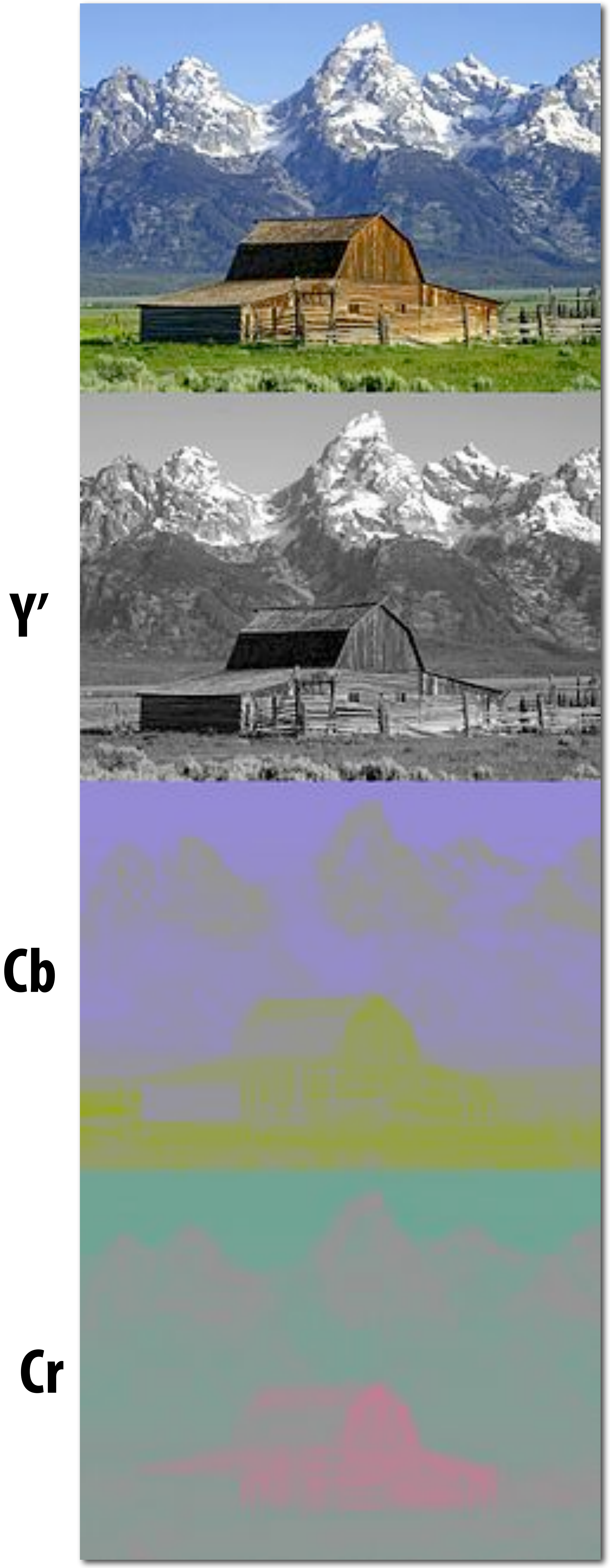
Another basis: Y'CbCr color space

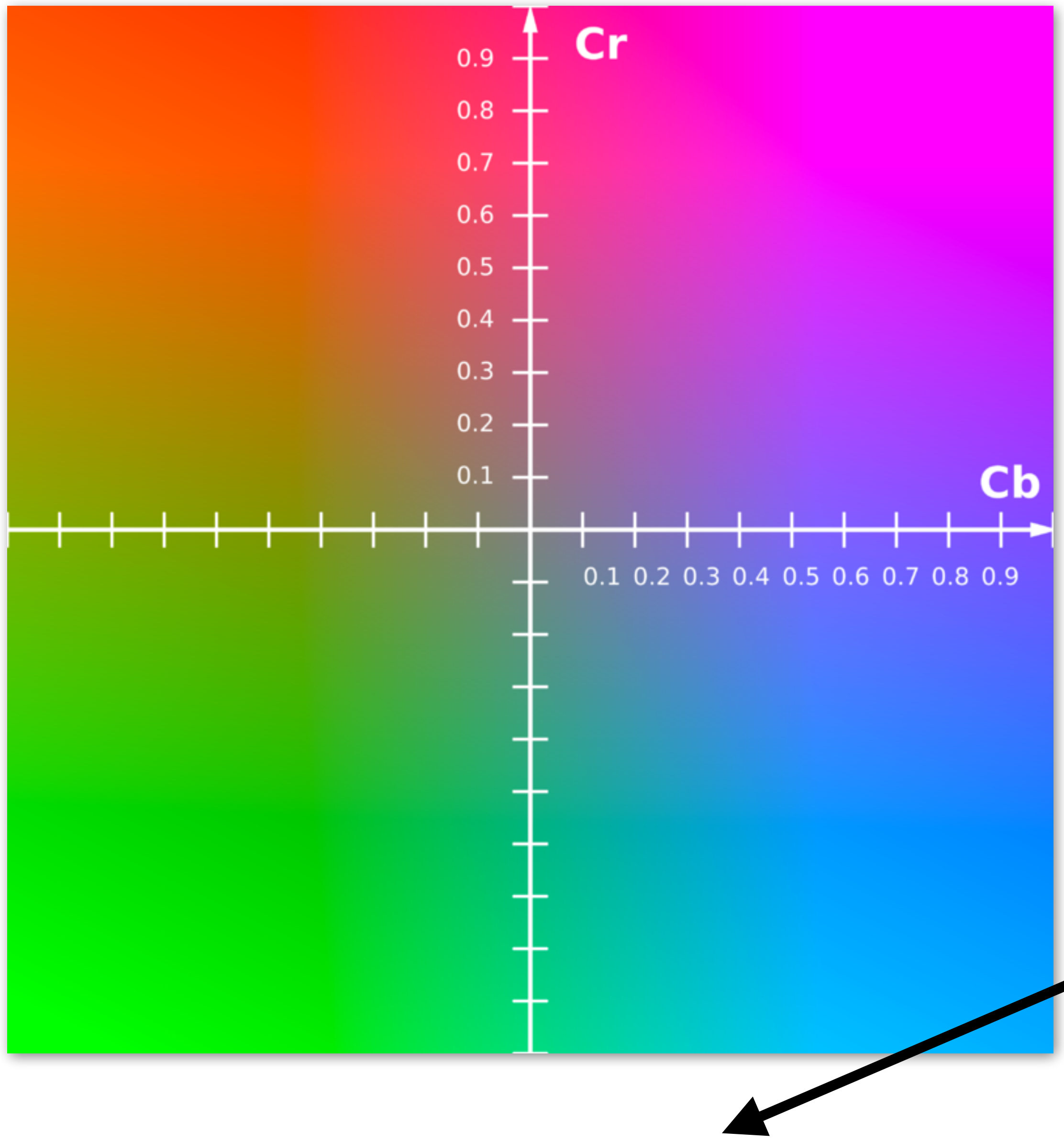
Recall: color is represented as a 3D value

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

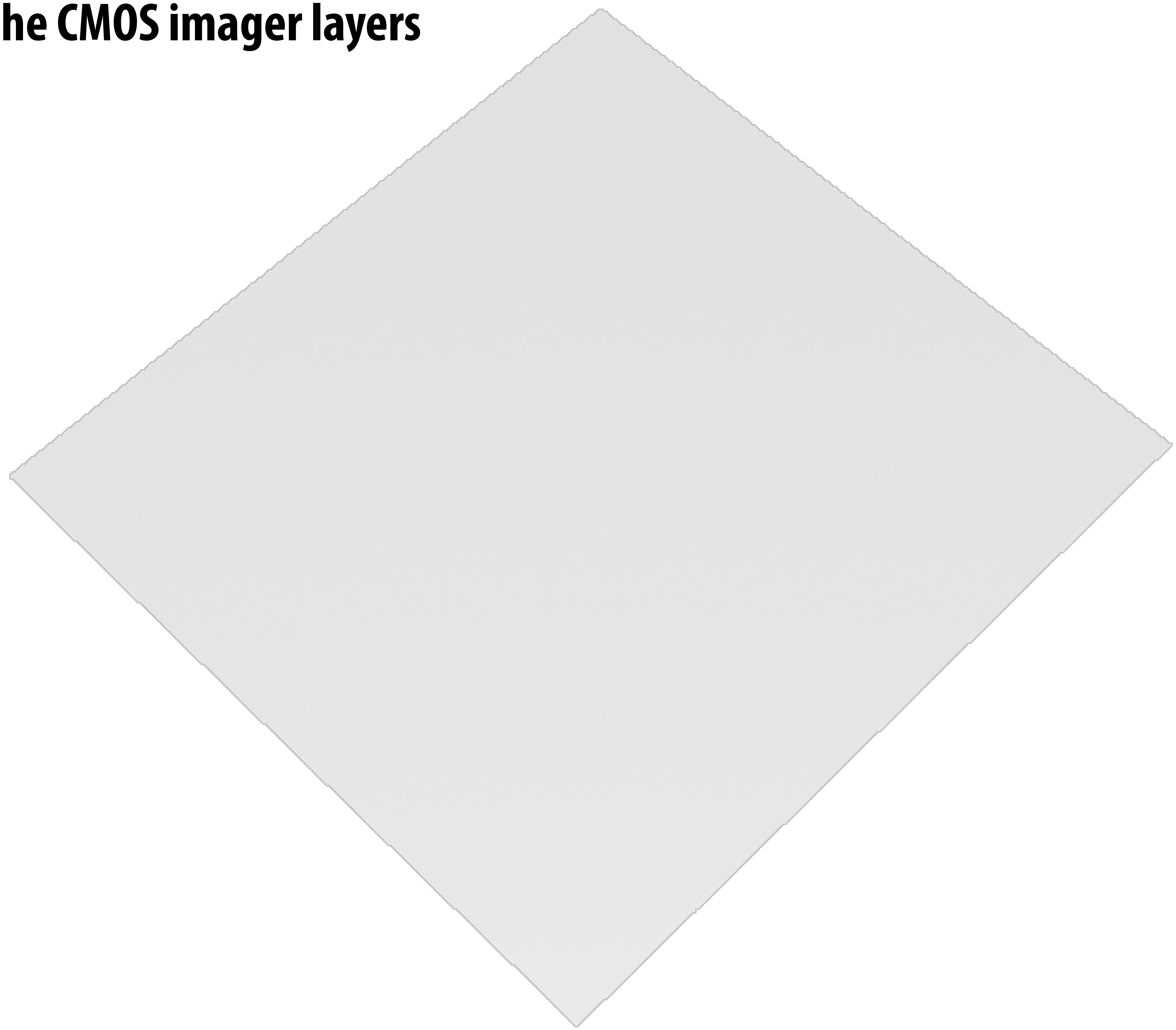
$$\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot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

Image credit: Wikipedia

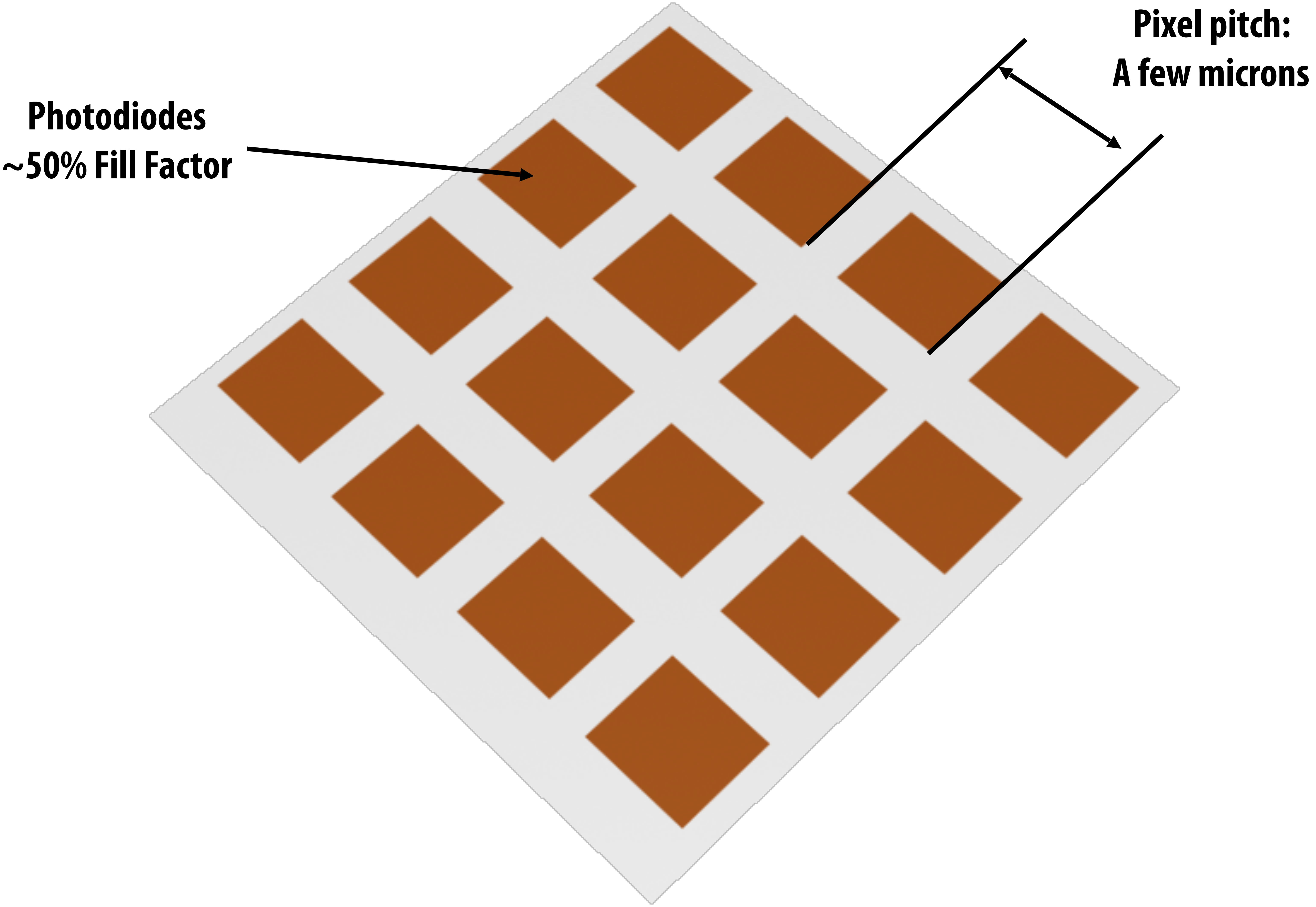
CMOS Pixel Structure

Front-side-illuminated (FSI) CMOS

Building up the CMOS imager layers

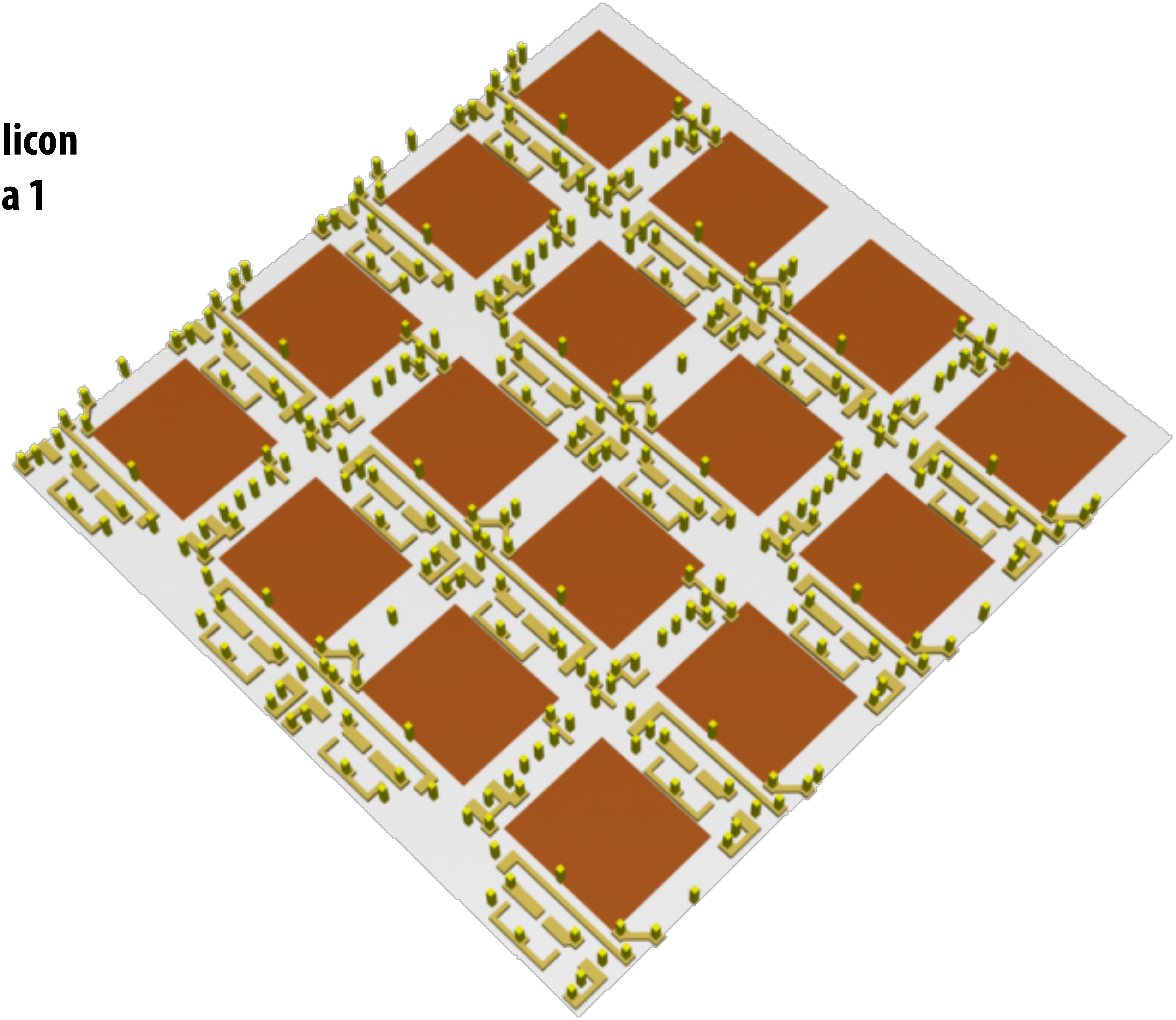


Courtesy R. Motta, Pixim



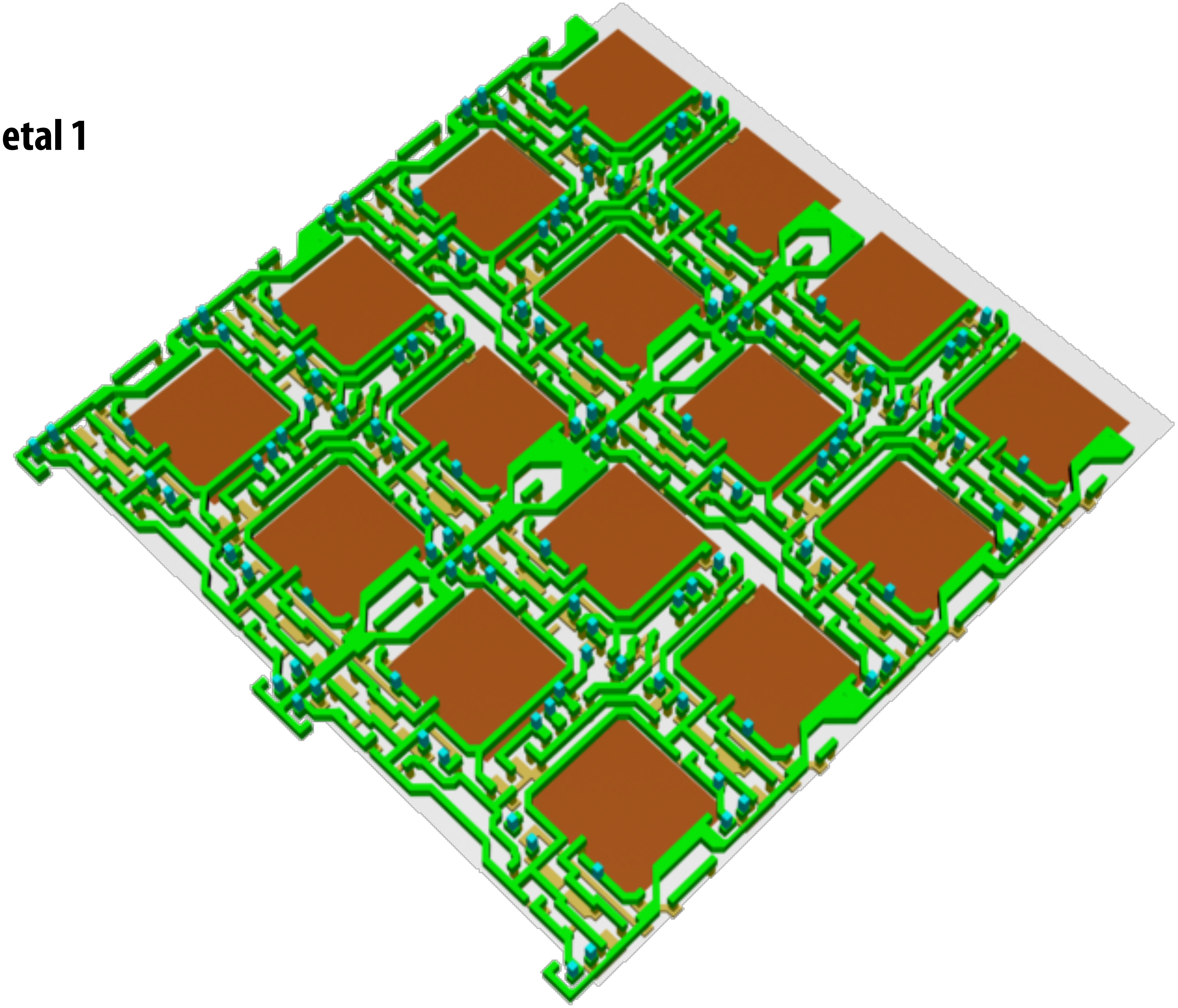
Courtesy R. Motta, Pixim

Polysilicon & Via 1



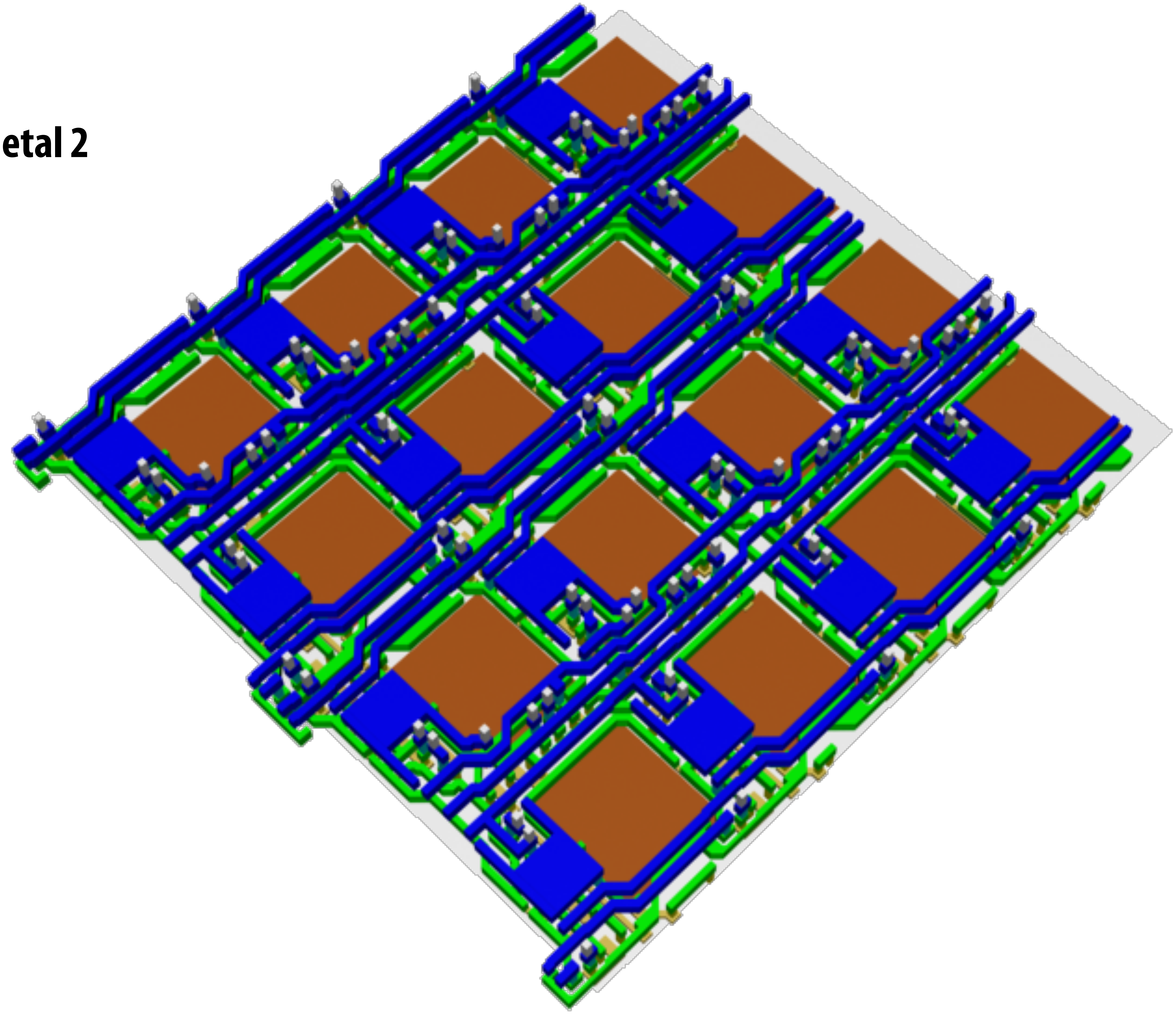
Courtesy R. Motta, Pixim

Metal 1



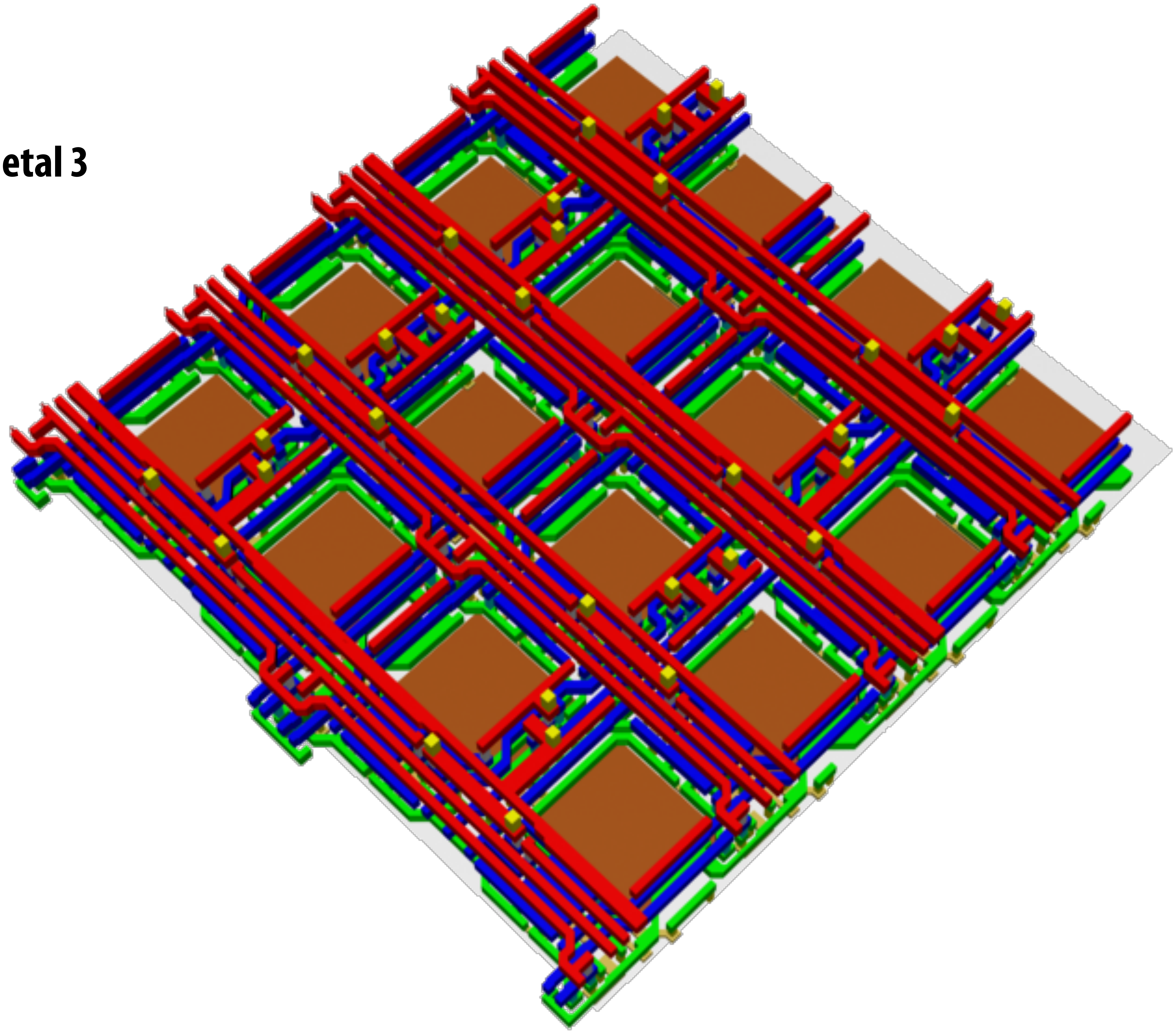
Courtesy R. Motta, Pixim

Metal 2



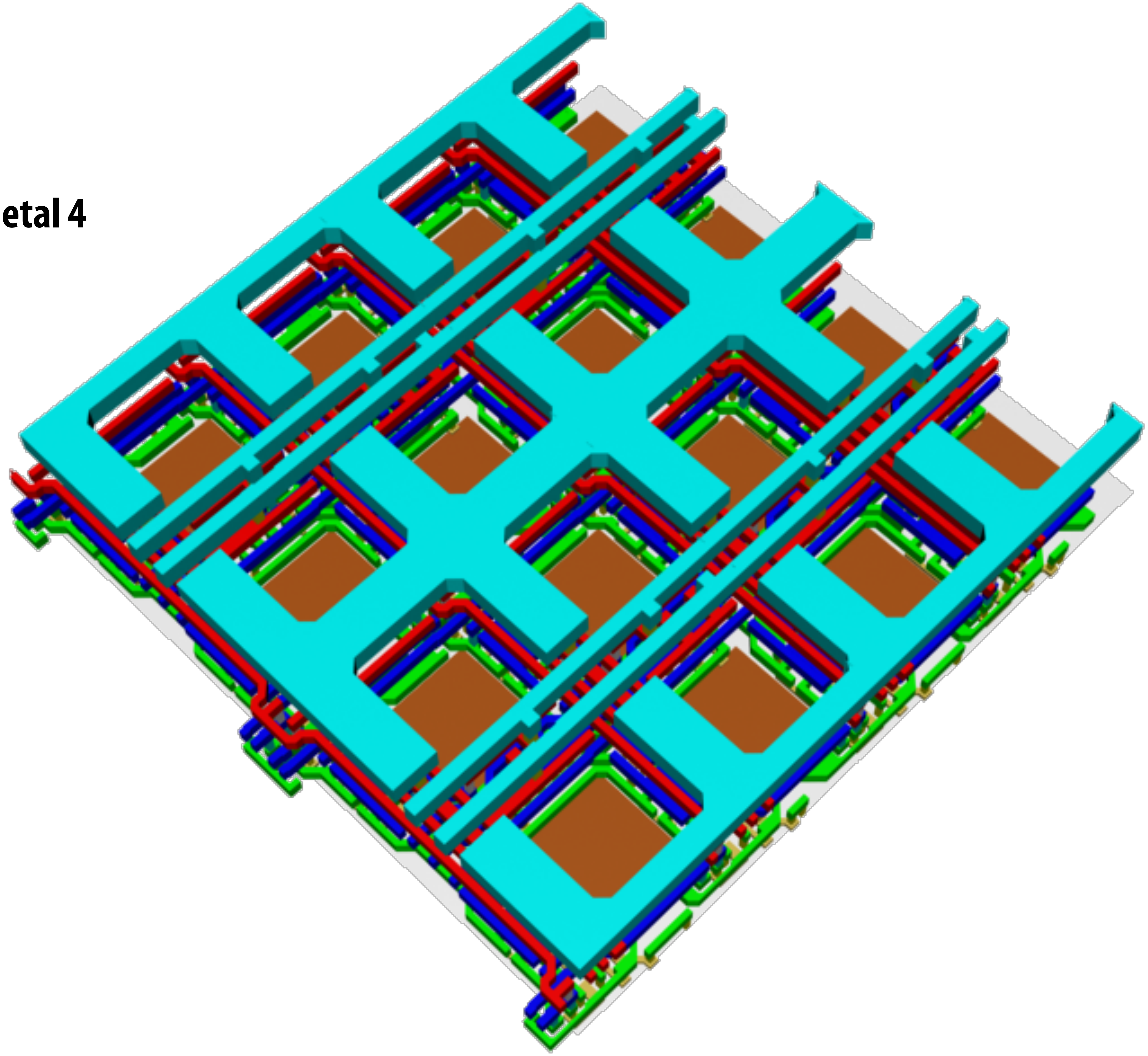
Courtesy R. Motta, Pixim

Metal 3



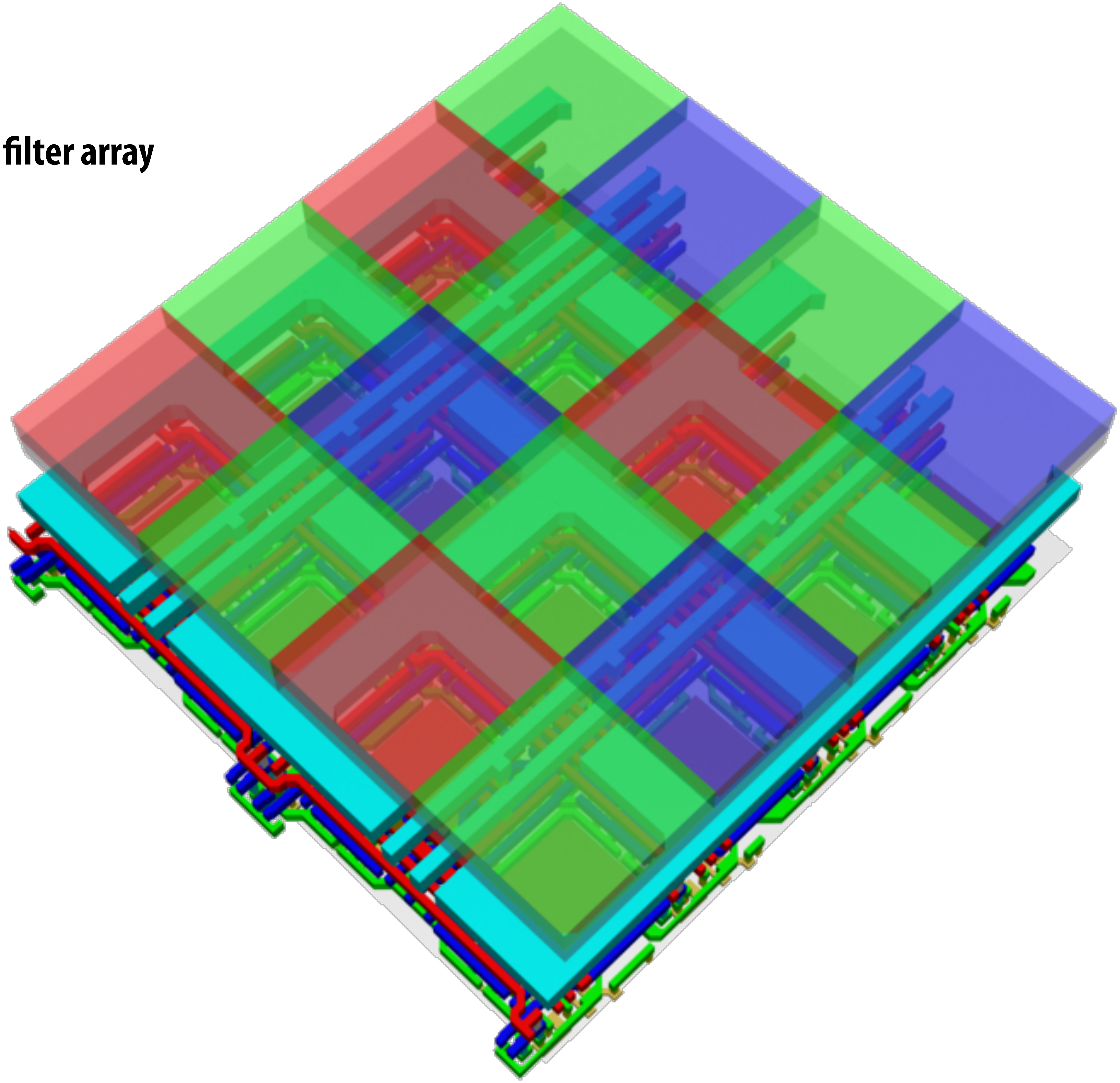
Courtesy R. Motta, Pixim

Metal 4



Courtesy R. Motta, Pixim

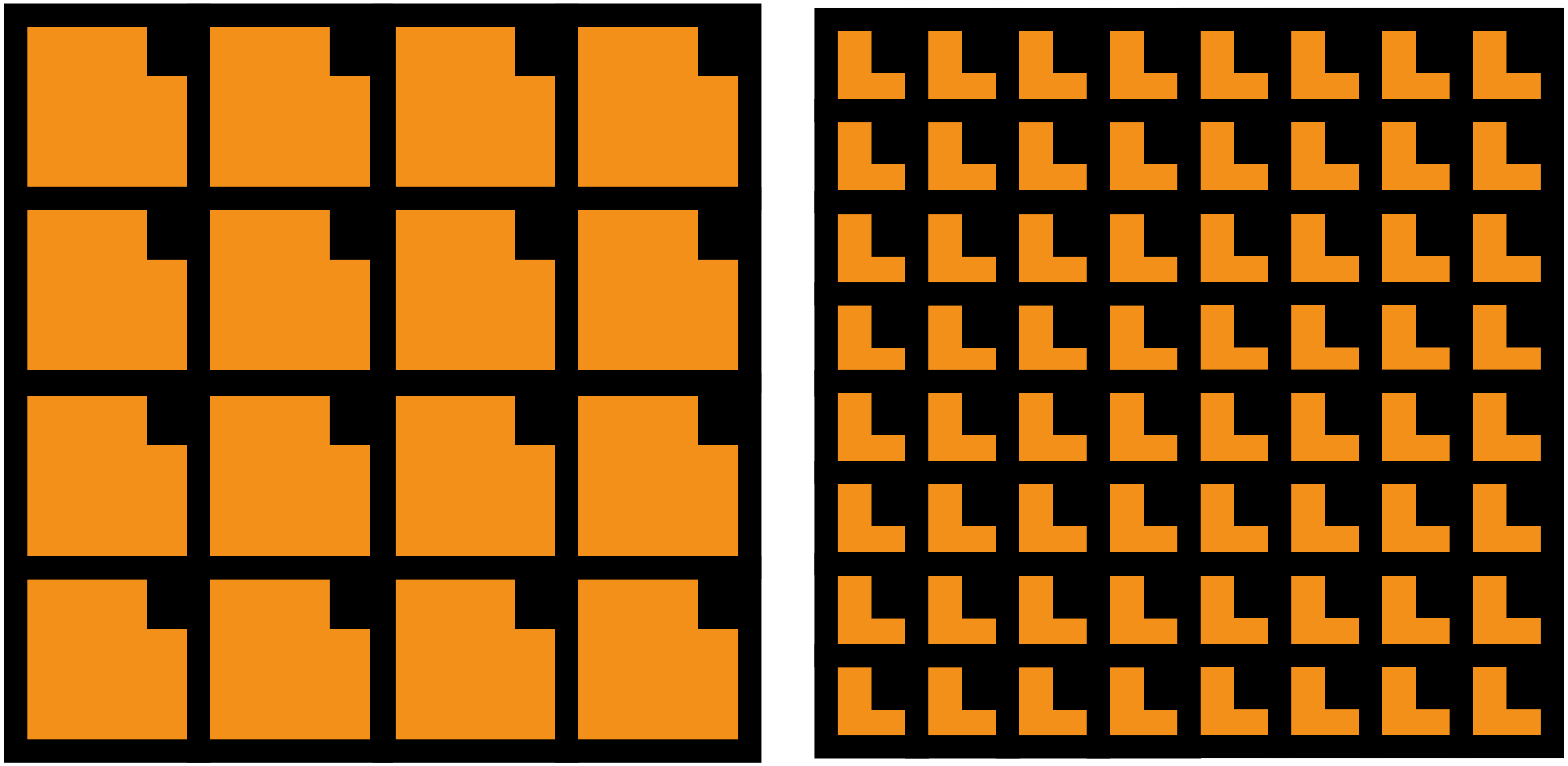
Color filter array



Courtesy R. Motta, Pixim

Pixel fill factor

Fraction of pixel area that integrates incoming light

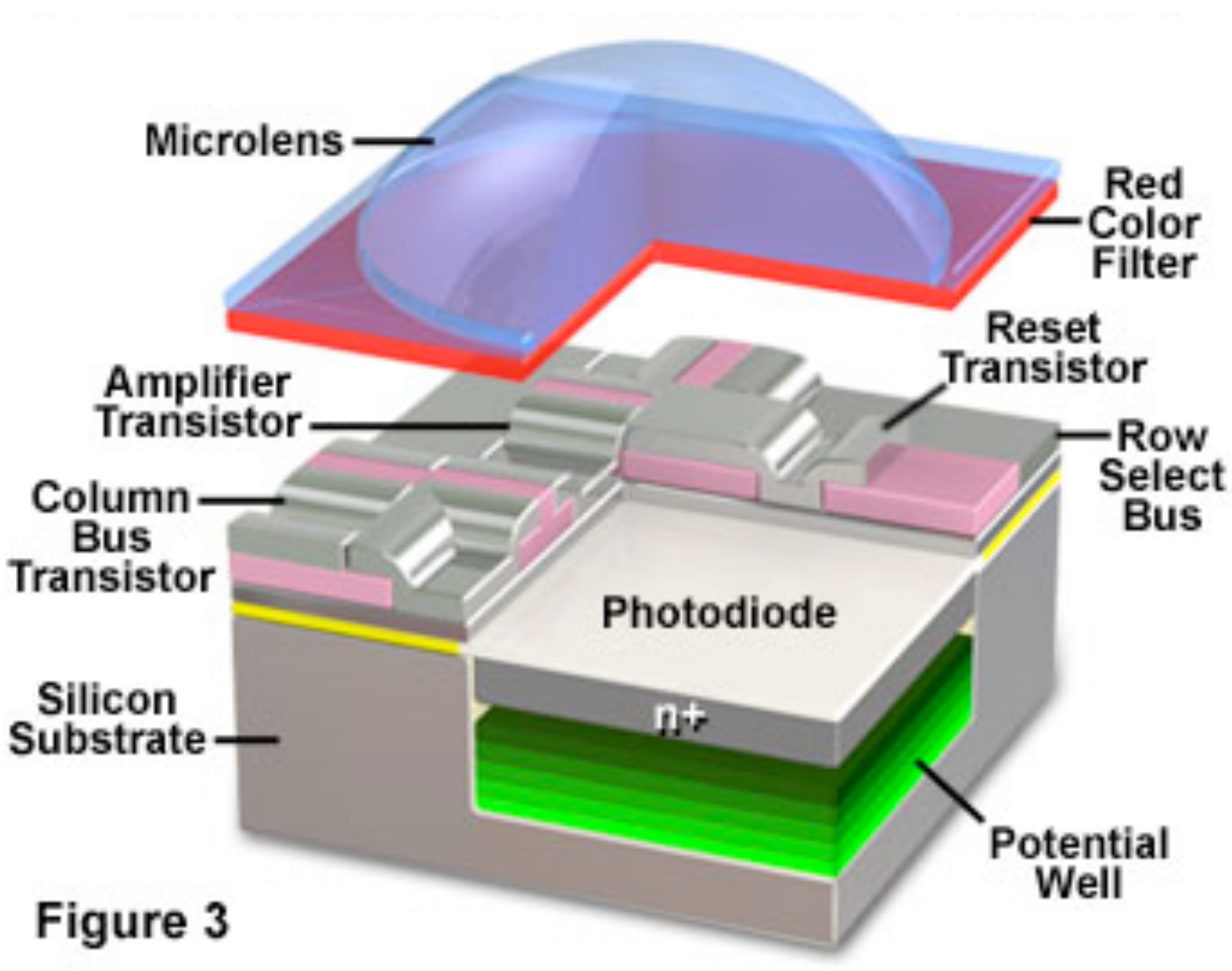


🟧 Photodiode area

⬛ Non photosensitive (circuitry)

Slide credit: Ren Ng

CMOS sensor pixel



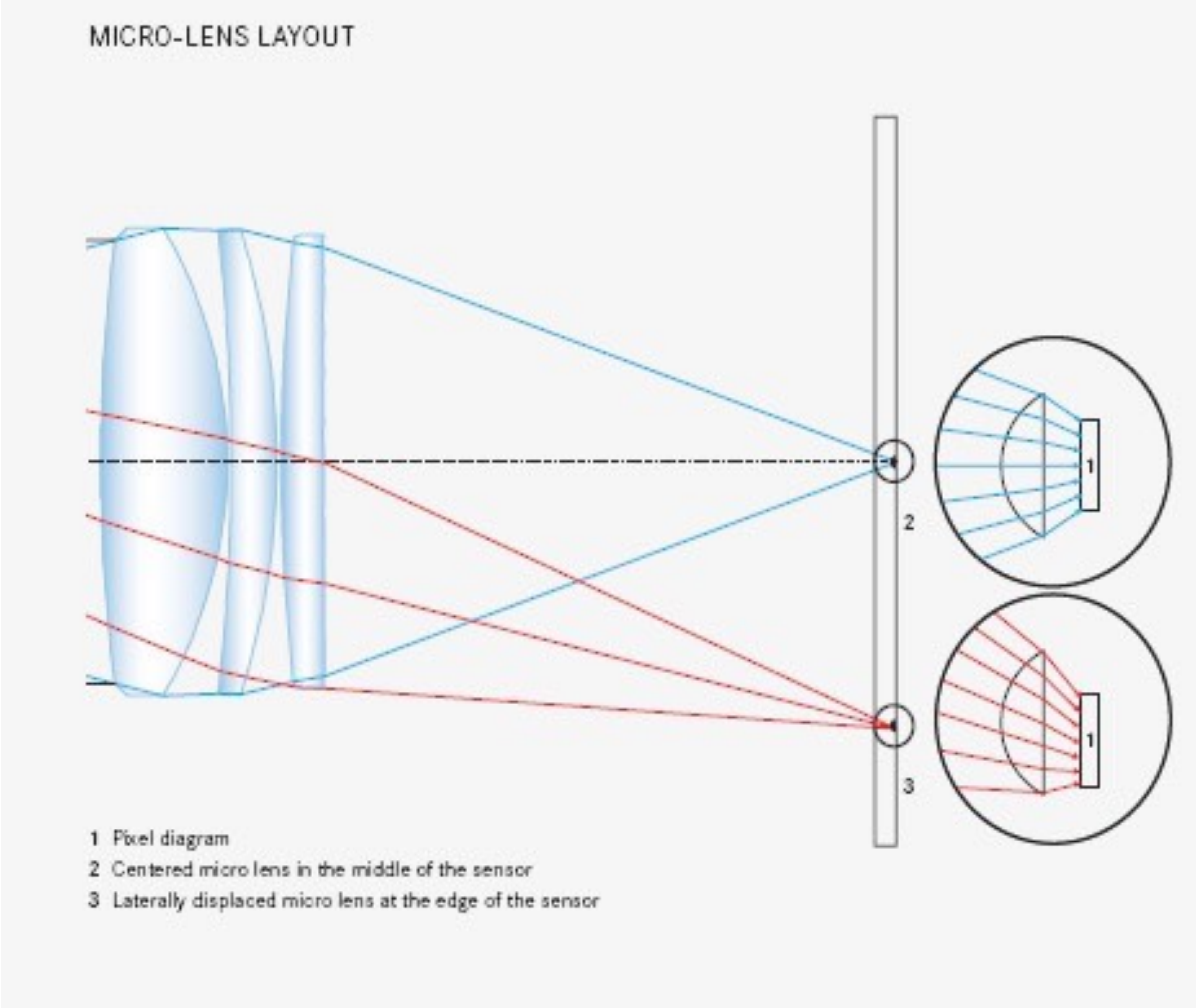
Color filter attenuates light

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

Microlens also serves to prefilter signal. Why?

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

Using micro lenses to improve fill factor

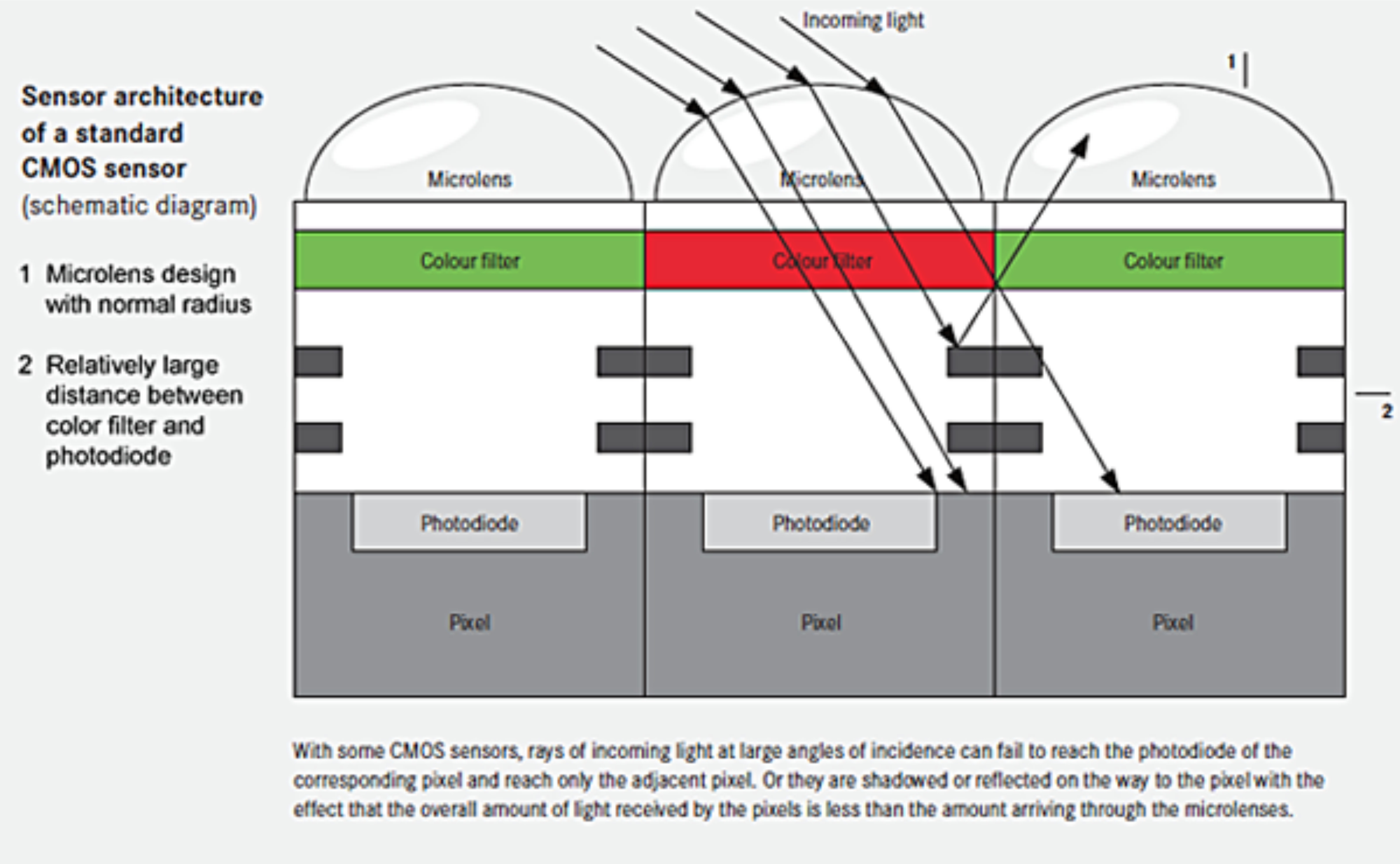

MICRO-LENS LAYOUT
1
2
1
3
1 Pixel diagram
2 Centered micro lens in the middle of the sensor
3 Laterally displaced micro lens at the edge of the sensor

Shifted microlenses on M9 sensor.

Leica M9

Slide credit: Ren Ng

Optical cross-talk



Sensor architecture of a standard CMOS sensor (schematic diagram)

1 Microlens design with normal radius

2 Relatively large distance between color filter and photodiode

With some CMOS sensors, rays of incoming light at large angles of incidence can fail to reach the photodiode of the corresponding pixel and reach only the adjacent pixel. Or they are shadowed or reflected on the way to the pixel with the effect that the overall amount of light received by the pixels is less than the amount arriving through the microlenses.

Slide credit: Ren Ng

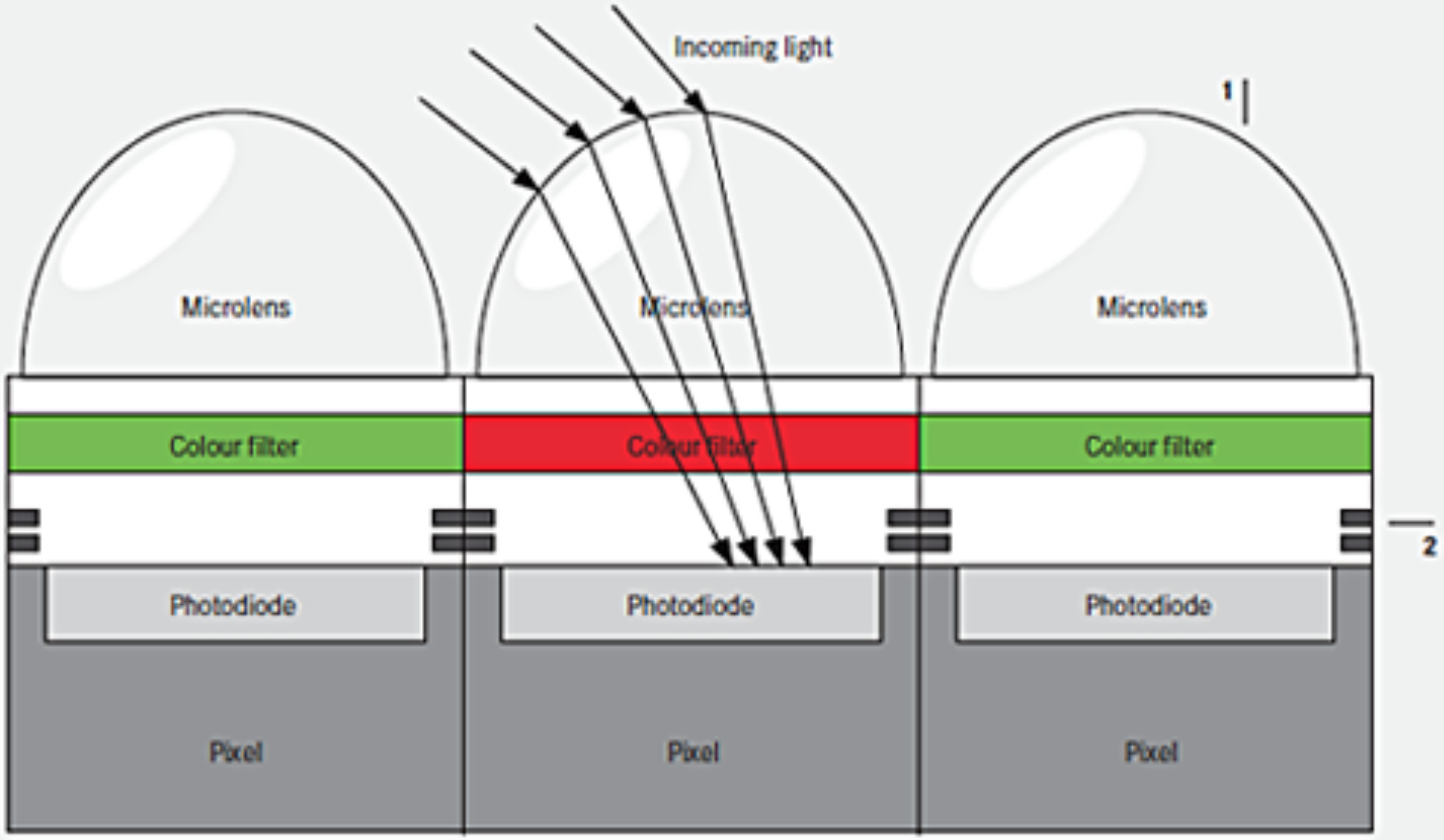
http://gmpphoto.blogspot.com/2012/09/the-new-leica-max-24mp-cmos-sensor.html

Pixel optics for minimizing cross-talk

Sensor architecture of the Leica Max 24 MP sensor (schematic diagram)

1 Microlens design with varying radius

2 Relatively short distance between color filter and photodiode



In the case of the Leica Max 24 MP sensor, and in contrast to standard CMOS sensors, even light rays with large angles of incidence, e.g. from wide-angle lenses or large apertures, are captured precisely by the photodiodes of the sensor. This is enabled by the special microlens design and the smaller distance between the colour filter and photodiode, which allows more light to enter the system, and ensures that it falls more directly on the respective photodiodes.

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)

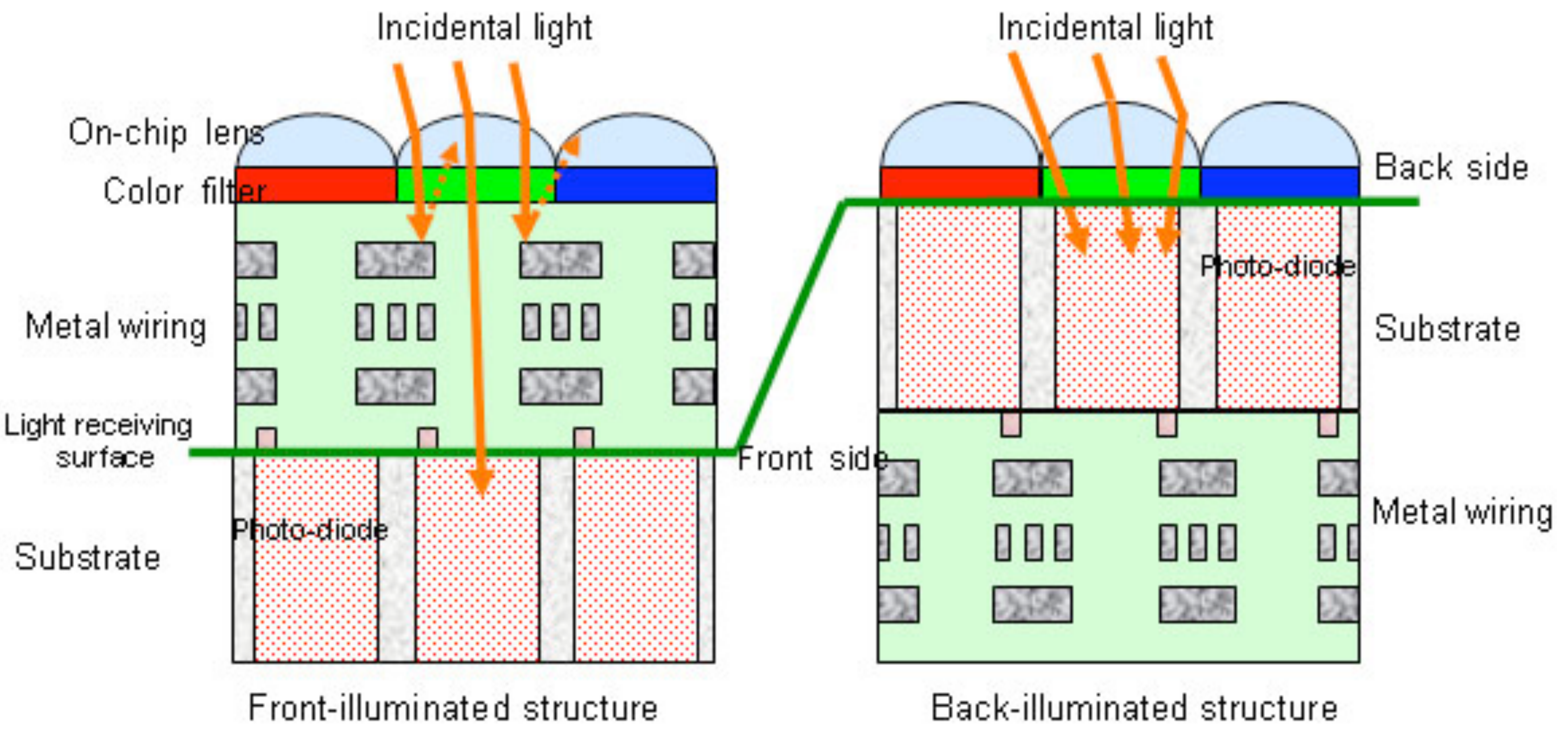
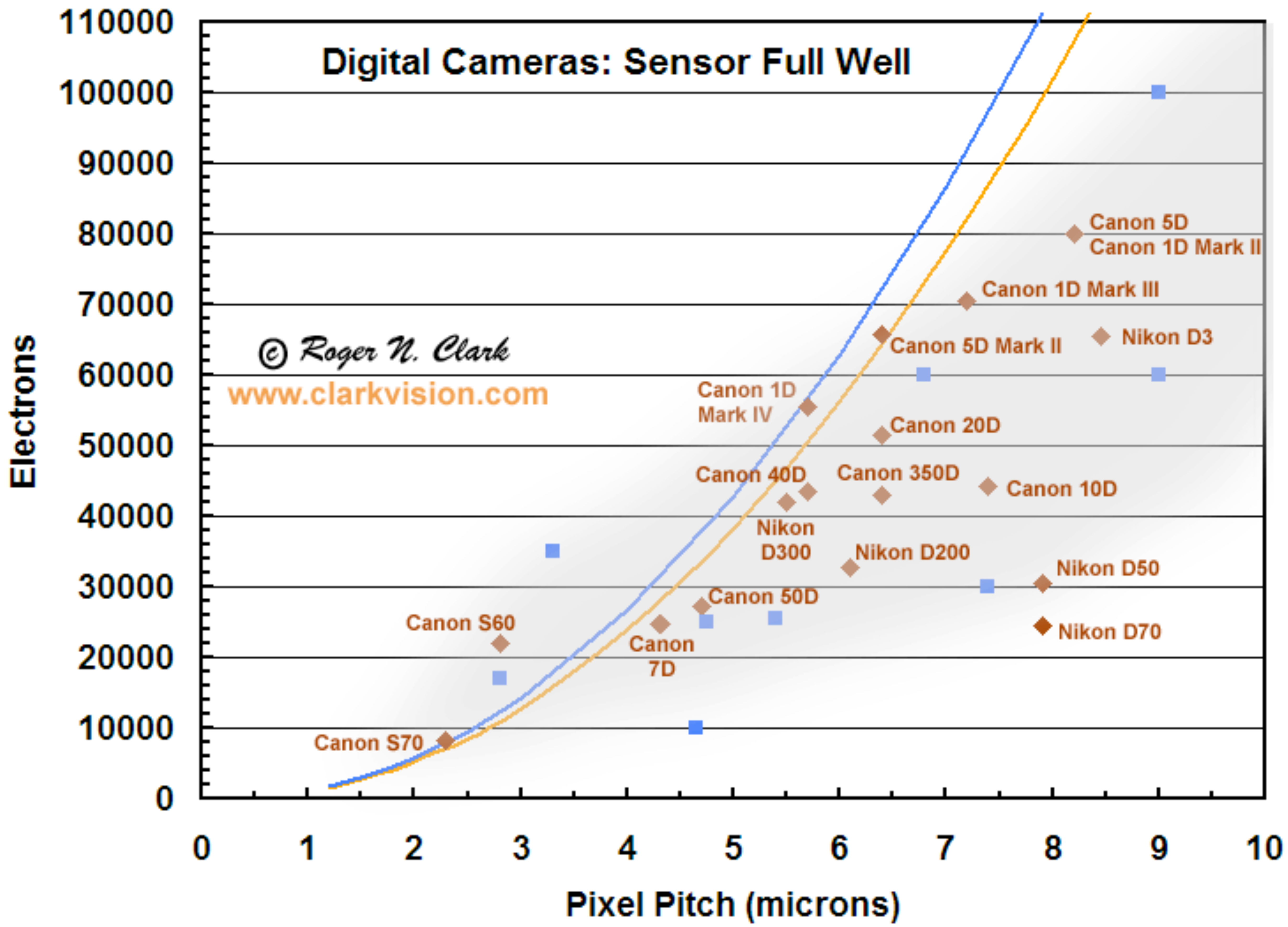


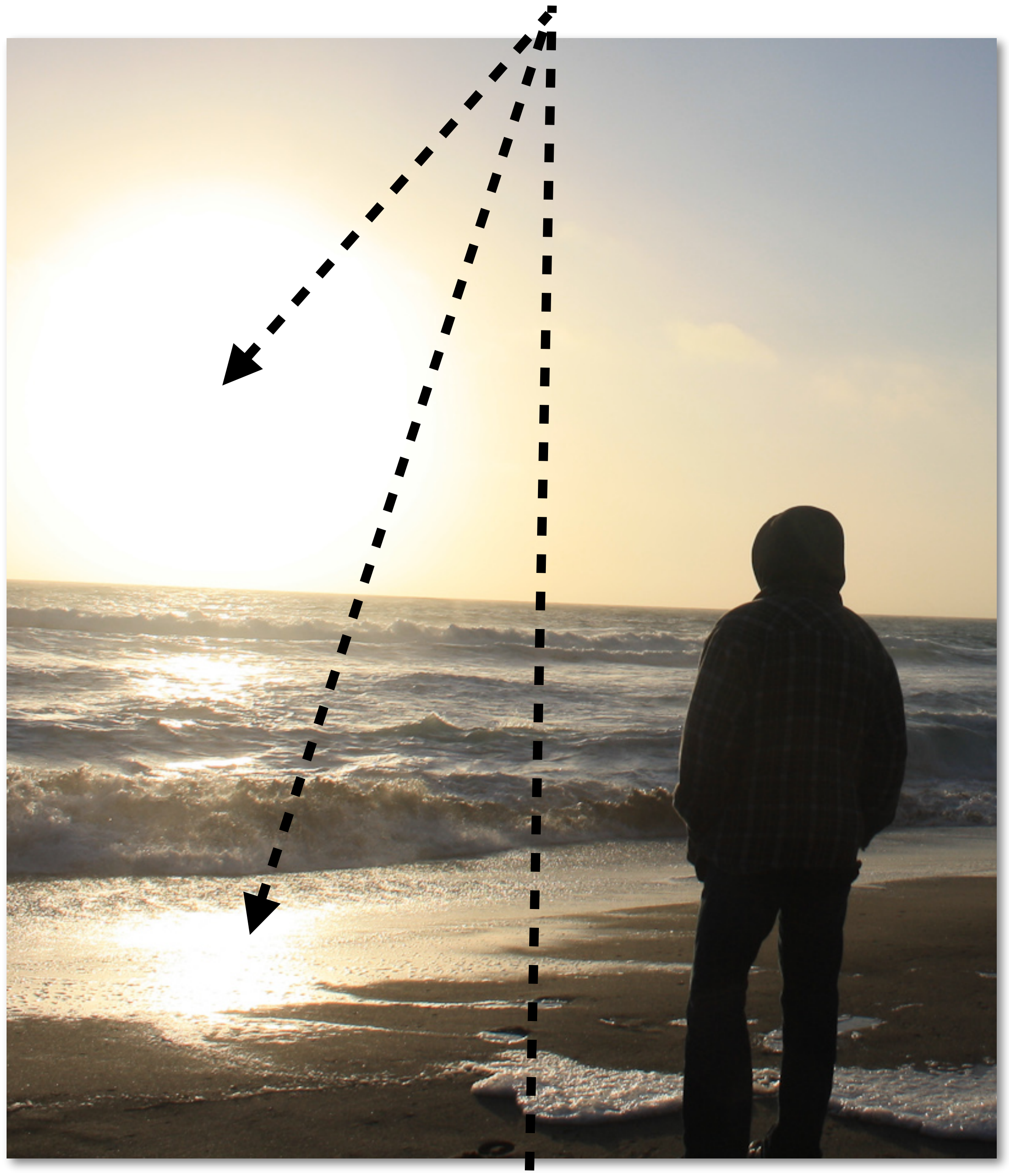
Illustration credit: Sony

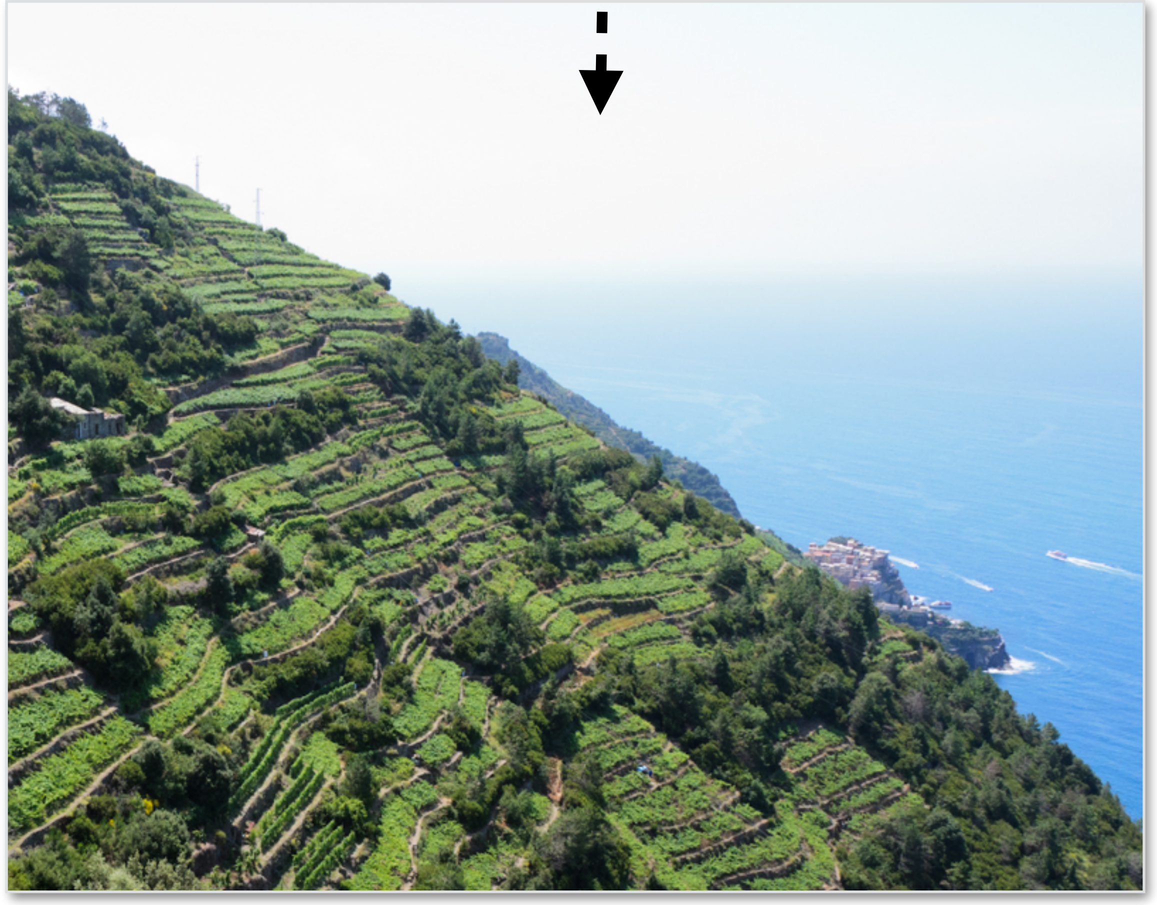
Full-well capacity

Pixel saturates when photon capacity is exceeded



Oversaturated pixels

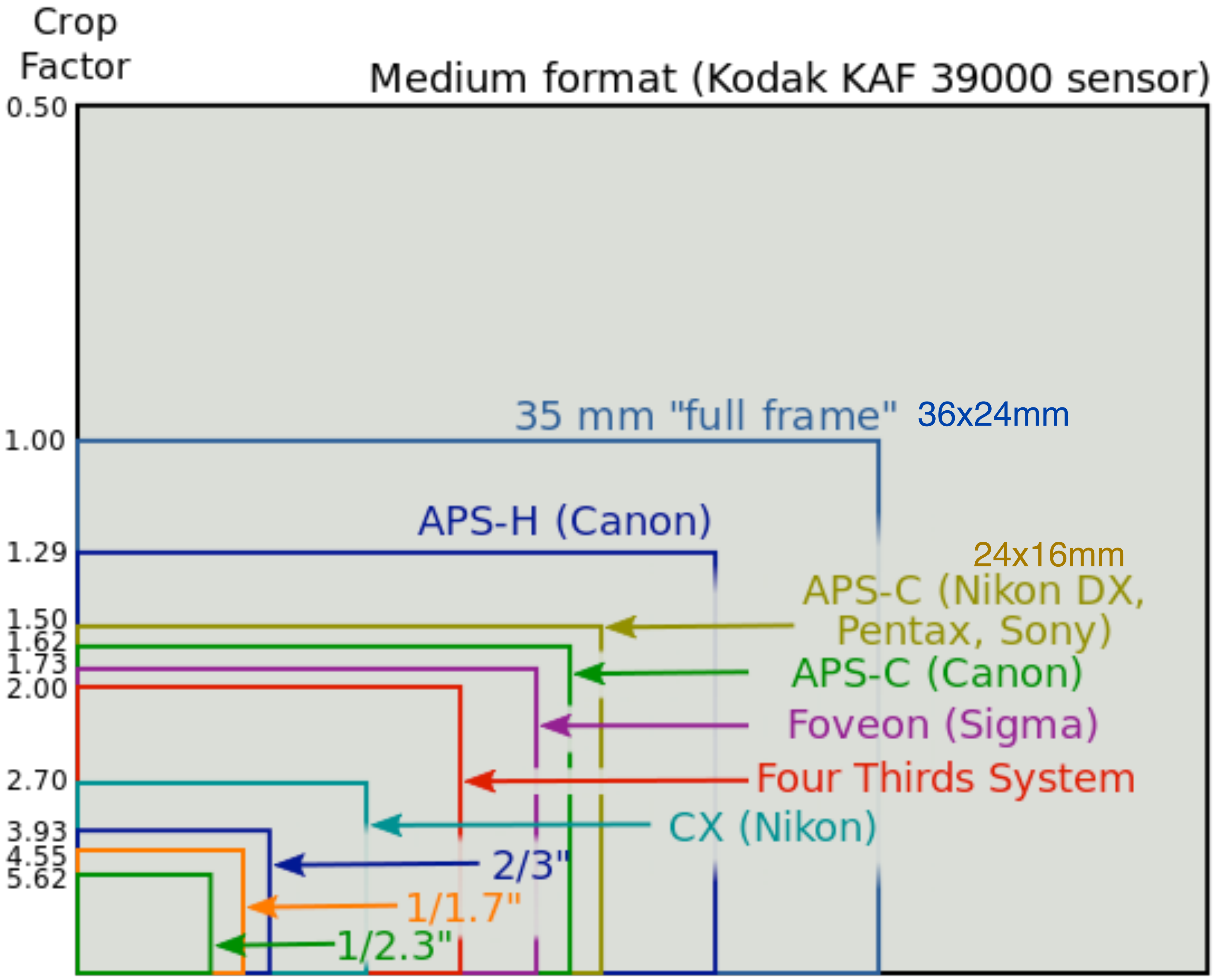


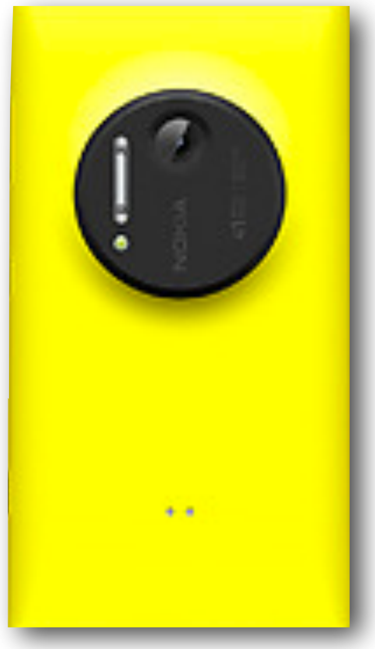


Graph credit: clarkvision.com

Bigger sensors = bigger pixels (or more pixels?)

- iPhone 6s (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 6s pixel size ~ 600 MP sensor





Nokia Lumia
(41 MP)

Image credit: Wikipedia

Steps in capturing an image

1. Clear sensor pixels
2. Open camera's mechanical shutter (exposure begins)
3. Optional: fire flash
4. Close camera mechanical shutter (exposure ends)
5. Read measurements off sensor
    - For each row:
        - Select row, read pixel for all columns in parallel
        - Pass data stream through amplifier and ADC

Electronic rolling shutter

Many cameras do not have a mechanical shutter (e.g., smart-phone cameras)

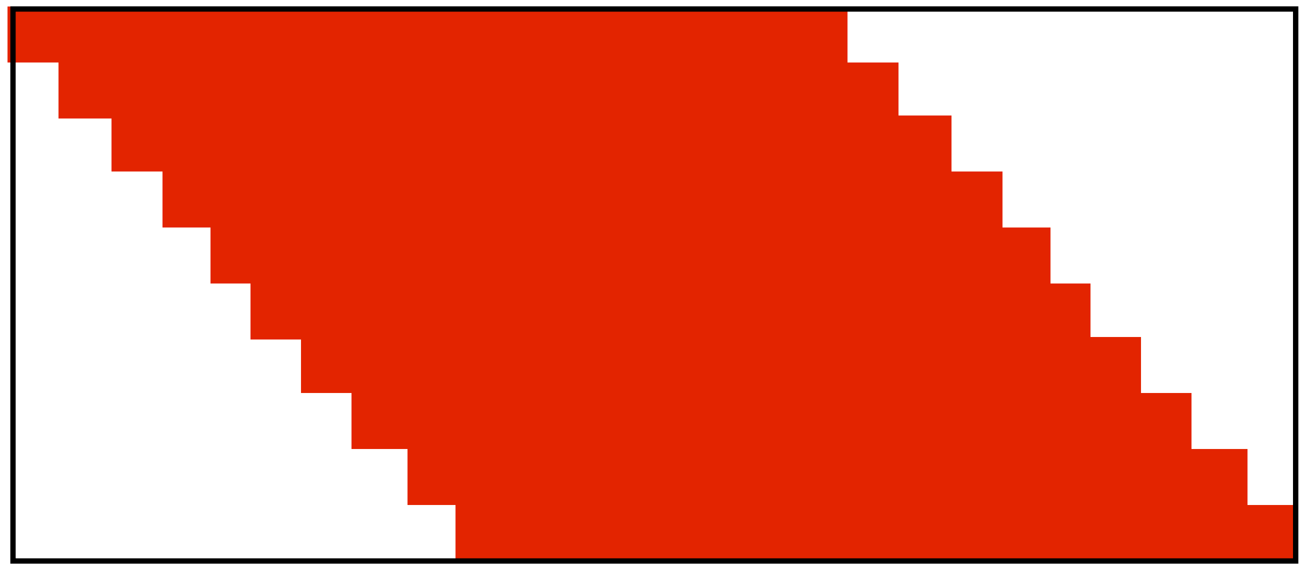
Exposure

1. Clear sensor pixels for row i (exposure begins)
2. Clear sensor pixels for row i+1 (exposure begins)

...

3. Read row i (exposure ends)
4. Read row i+1 (exposure ends)

Photo of red square, moving to right over duration sensor is exposed



Each image row exposed for the same amount of time (same exposure)

Each image row exposed over different interval of time (time offset determined by row read speed)

Rolling shutter effects

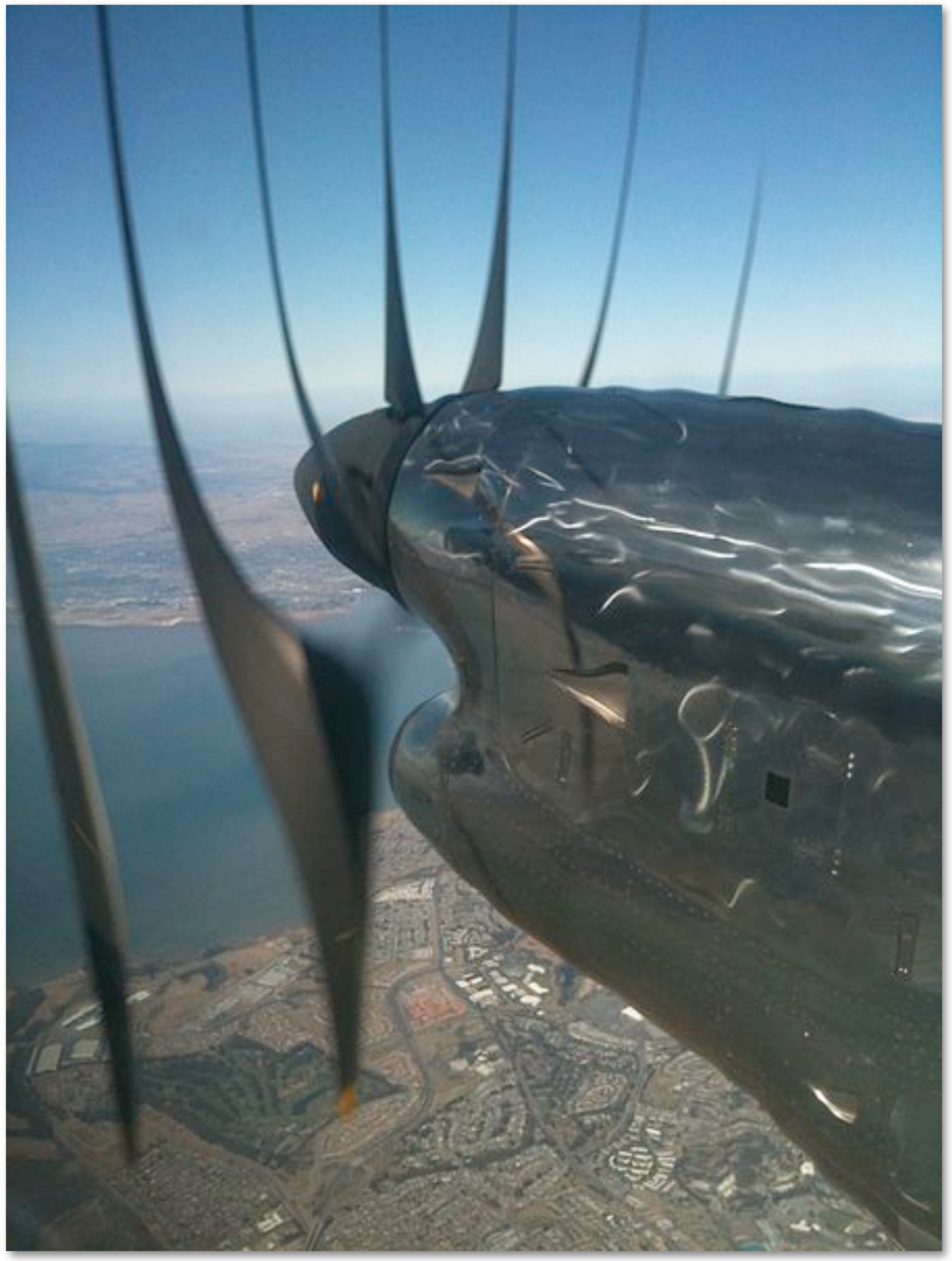


Image credit: Wikipedia

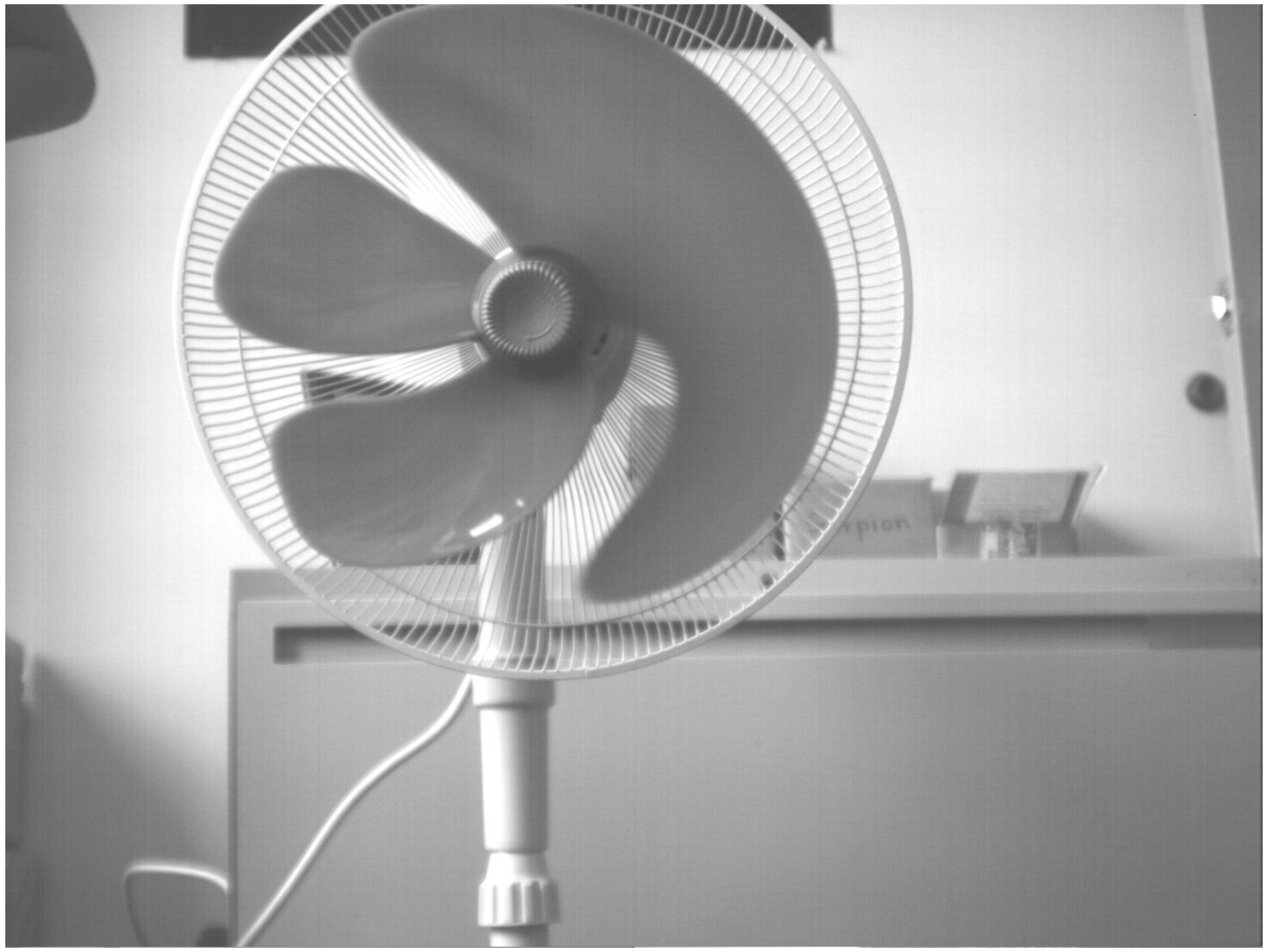
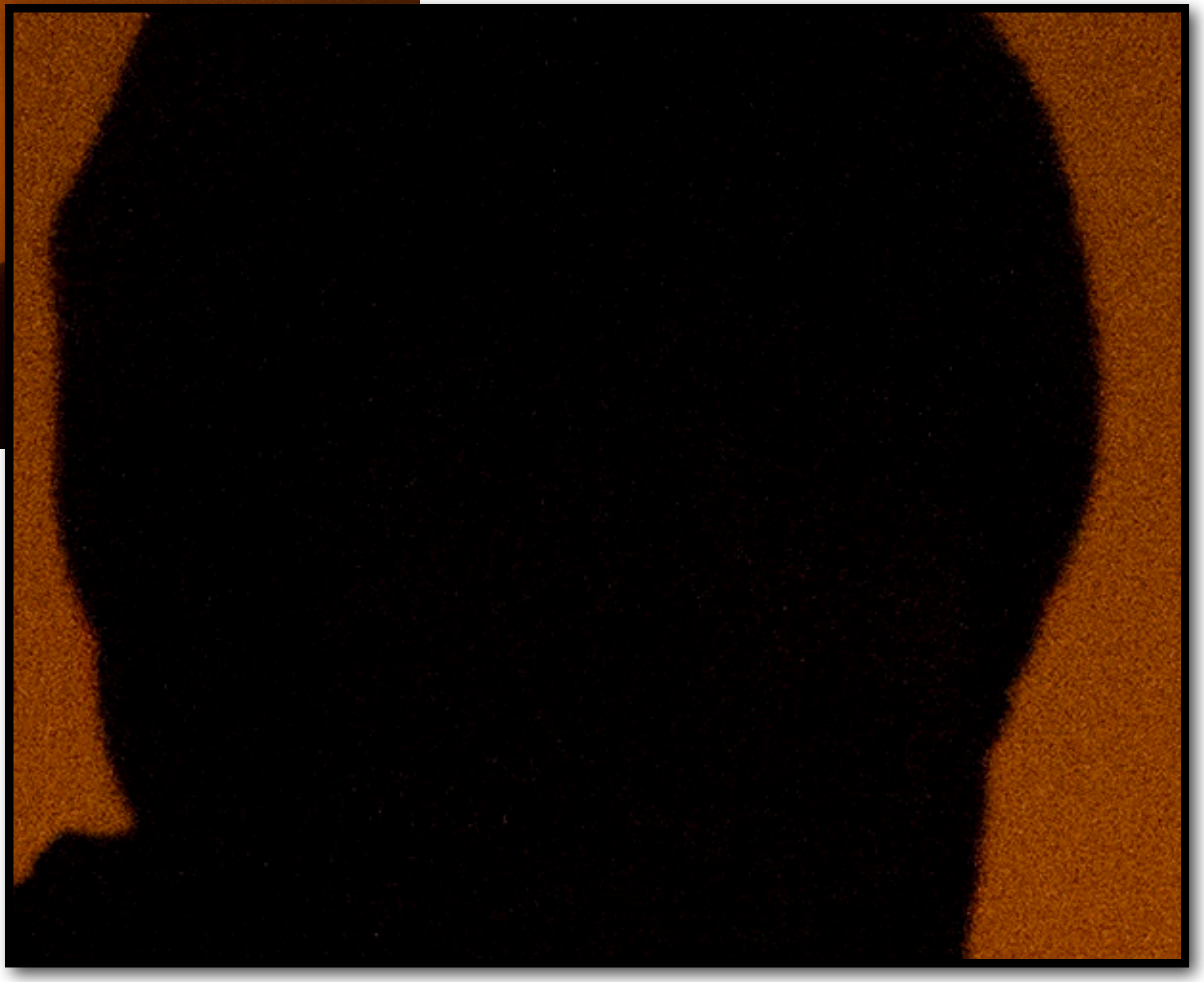


Image credit: Point Grey Research

Measurement noise



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

(or in shadows in daytime images)

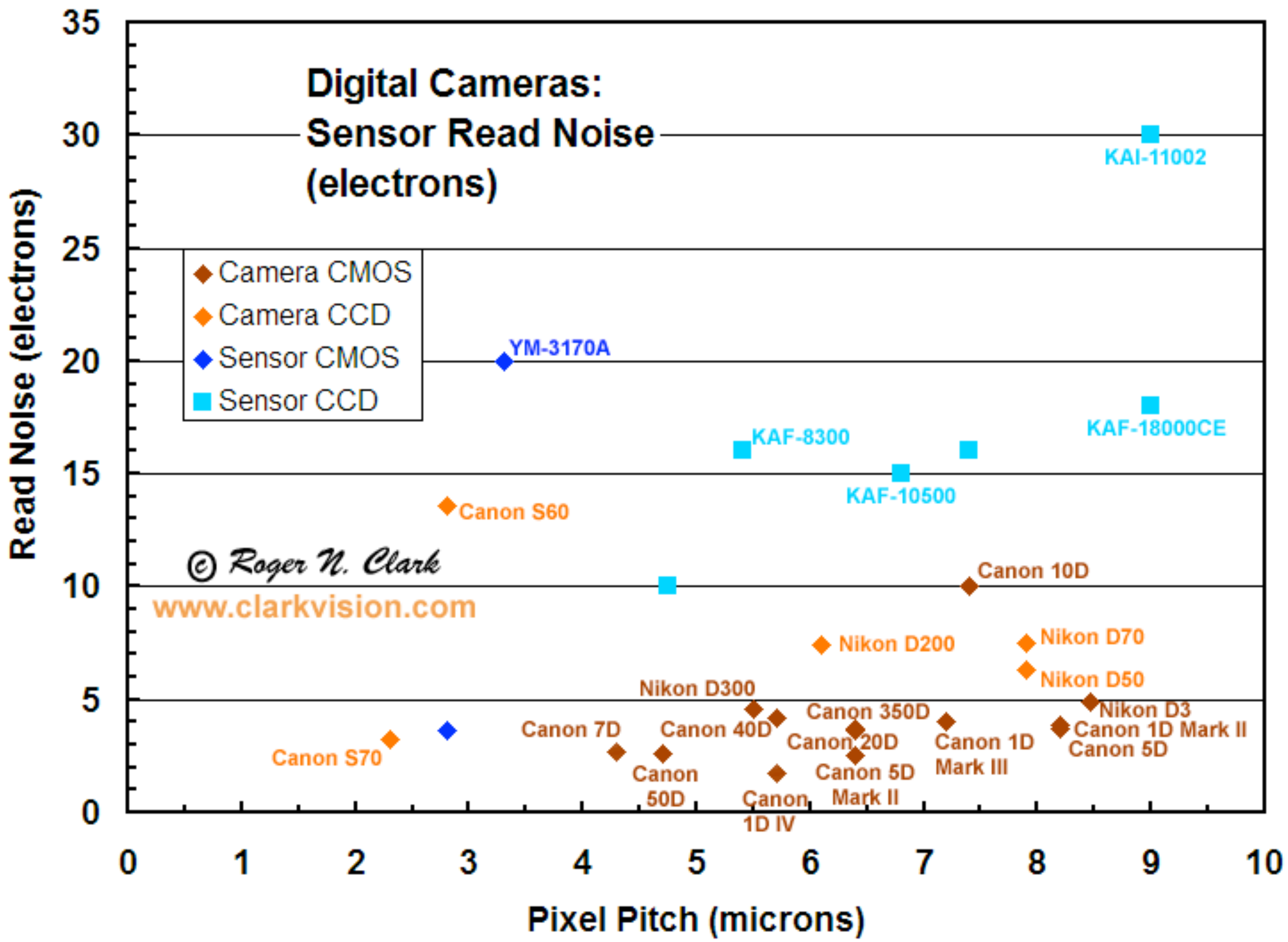
Measurement noise

- Photon shot noise:
  - Photon arrival rates feature poisson distribution
  - Standard deviation = sqrt(N)
  - Signal-to-noise ratio (SNR): N/sqrt(N)
  - 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)

Read noise



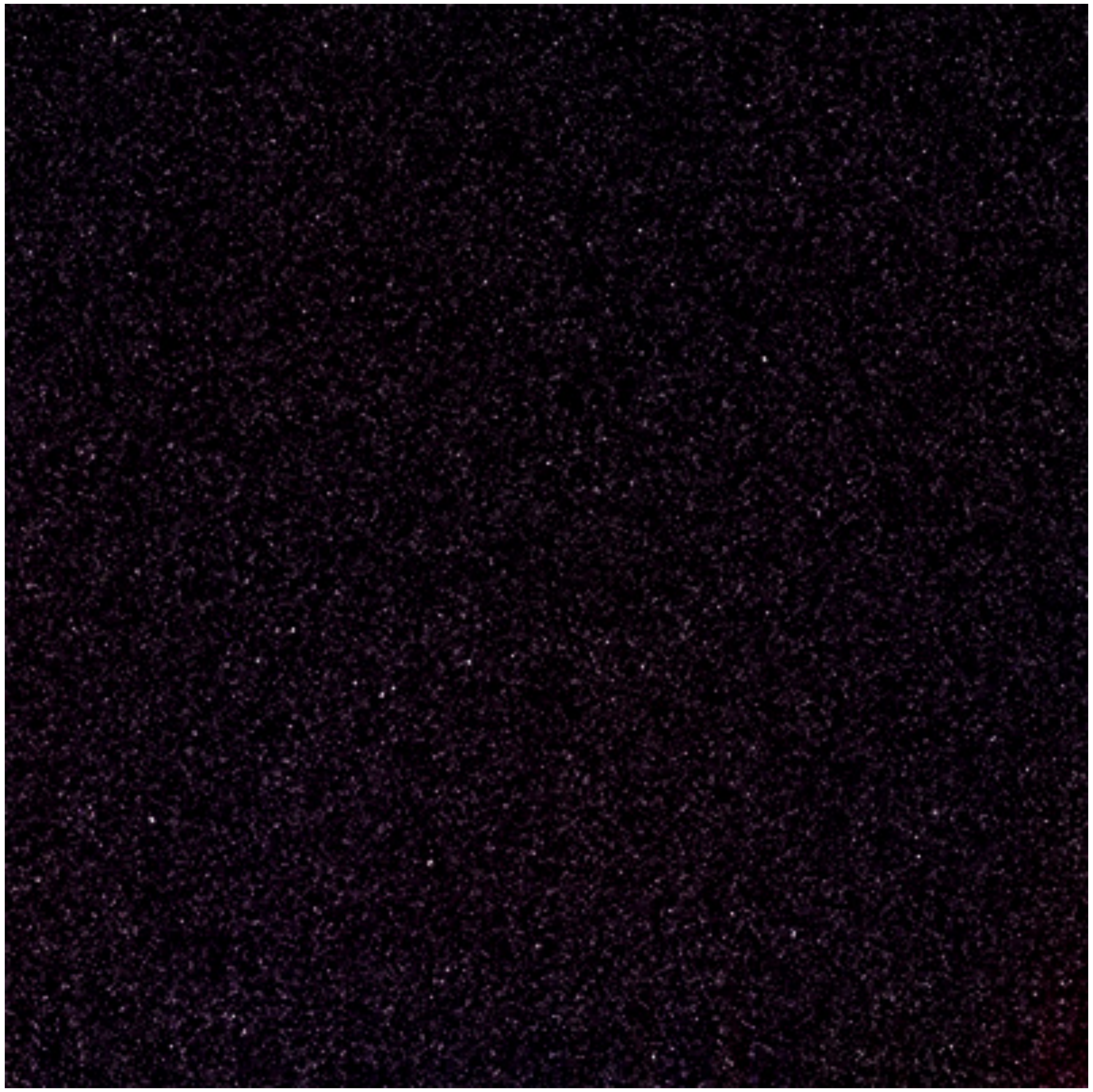
Read noise is largely independent of pixel size

Large pixels + bright scene: noise determined largely by photon shot noise

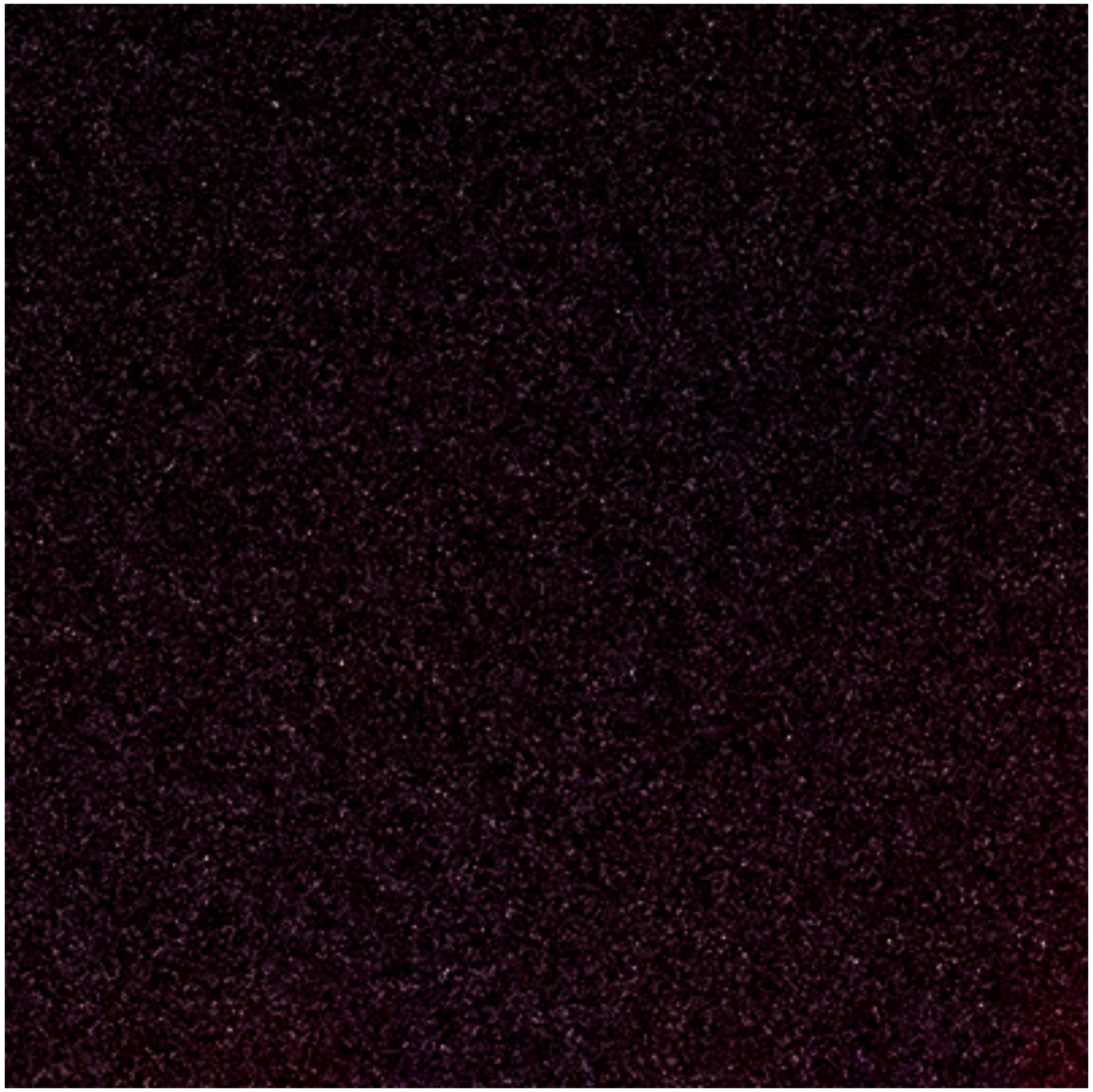
Image credit: clarkvision.com

Dark shot noise / read noise

Black image examples: Nikon D7000, High ISO



1/60 sec exposure



1 sec exposure

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 5s: 1.5 um

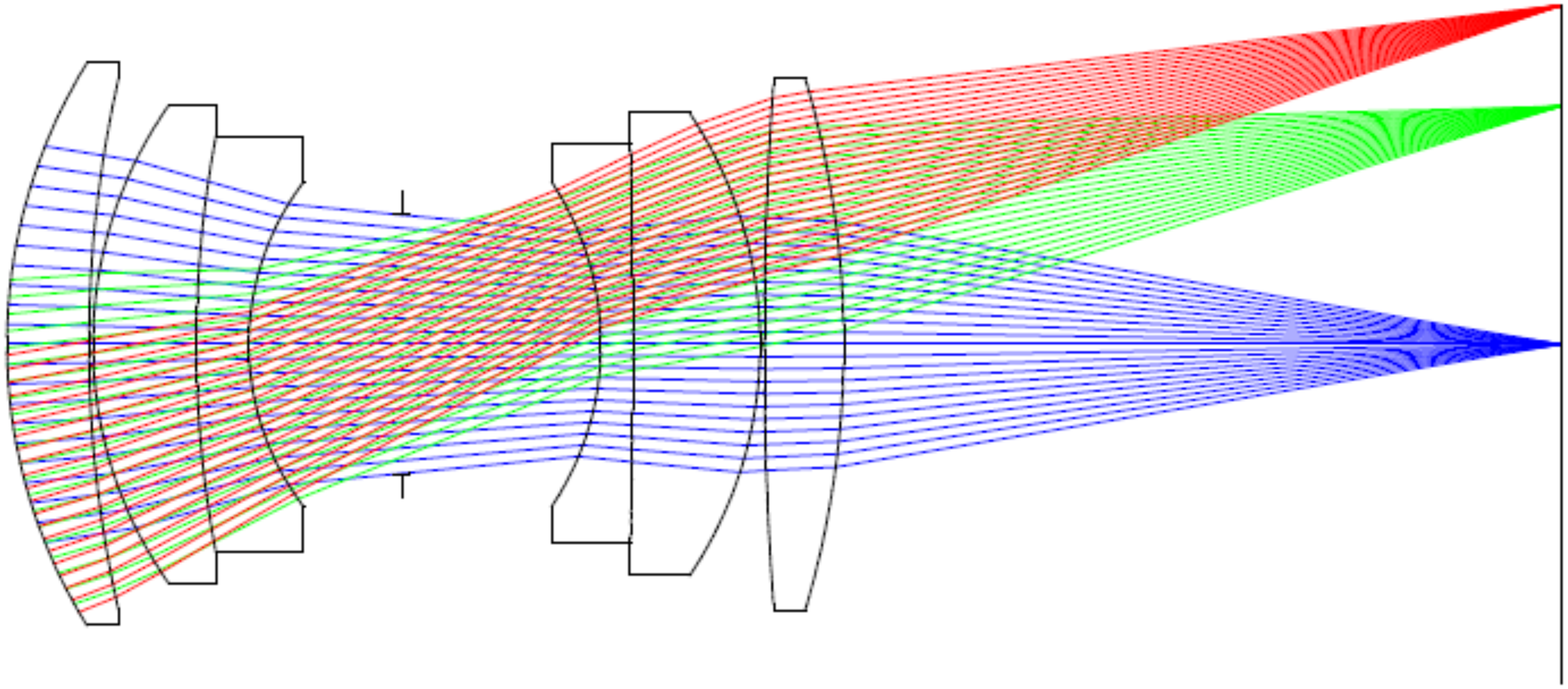
- **Sensitive pixels**
  - Good materials
  - High fill factor

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**

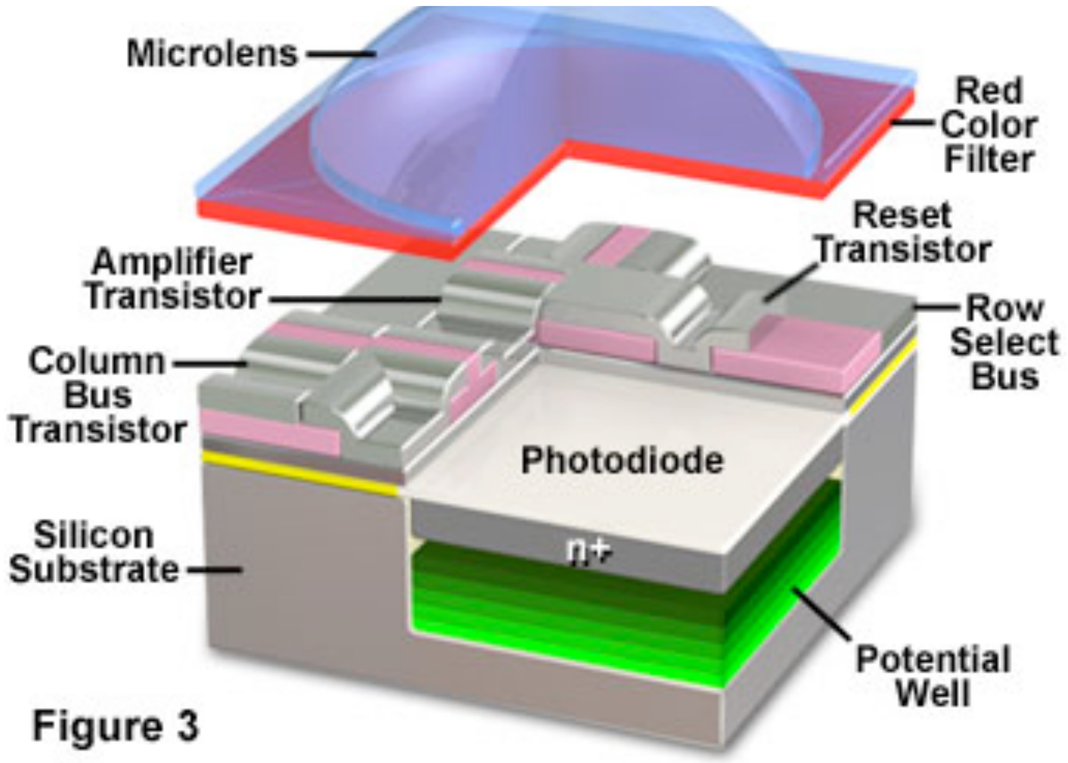
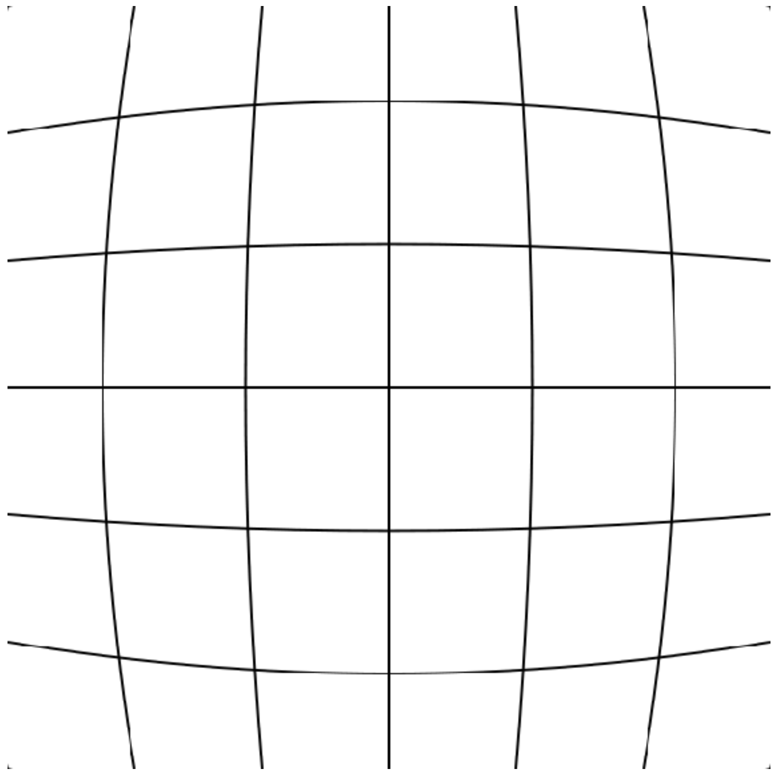


Image credit: Mark Butterworth

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!)
- **Dead pixels (stuck at white or black)**
- **Lens distortion**



Pincushion distortion

Captured Image

Corrected Image

Image credit: PCWorld

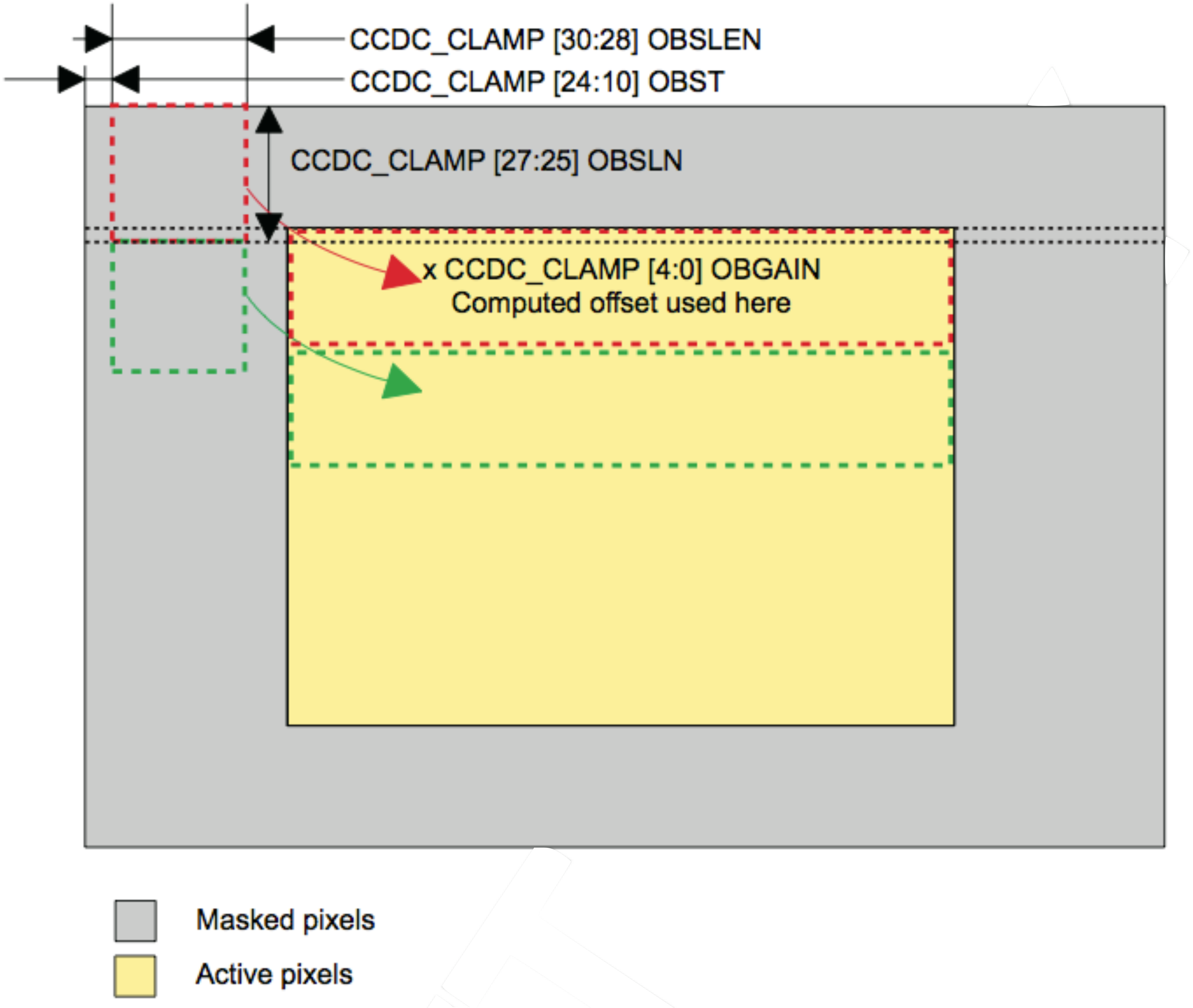
A simple RAW image processing pipeline

Adopting terminology from Texas Instruments OMAP Image Signal Processor pipeline
(since public documentation exists)

Assume: receiving 12 bits/pixel Bayer mosaiced data from sensor

Optical clamp: remove sensor offset bias

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



CCDC_CLAMP [30:28] OBSLEN
CCDC_CLAMP [24:10] OBST
CCDC_CLAMP [27:25] OBSLN
x CCDC_CLAMP [4:0] OBGAIN
Computed offset used here
Masked pixels
Active pixels

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

Step 2: 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;
```

Lens shading compensation

- **Correct for vignetting**
  - Recall 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

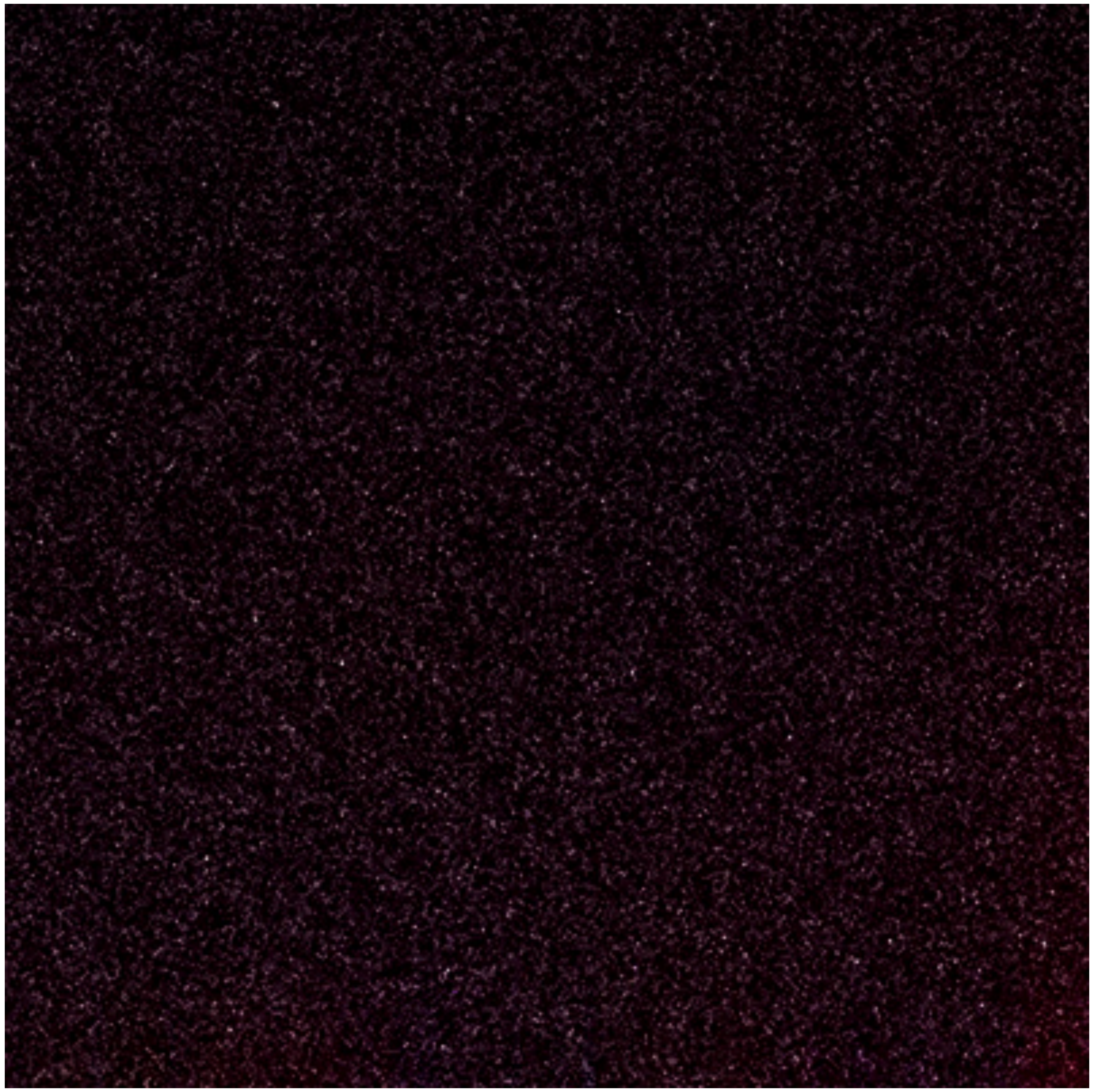
```
offset = upsample_compensation_offset_buffer(current_pixel_xy);
gain = upsample_compensation_gain_buffer(current_pixel_xy);

output_pixel = offset + gain * input_pixel;
```

Optional dark-frame subtraction

- **Similar computation to lens shading compensation**

```
output_pixel = input_pixel – dark_frame[current_pixel_xy];
```



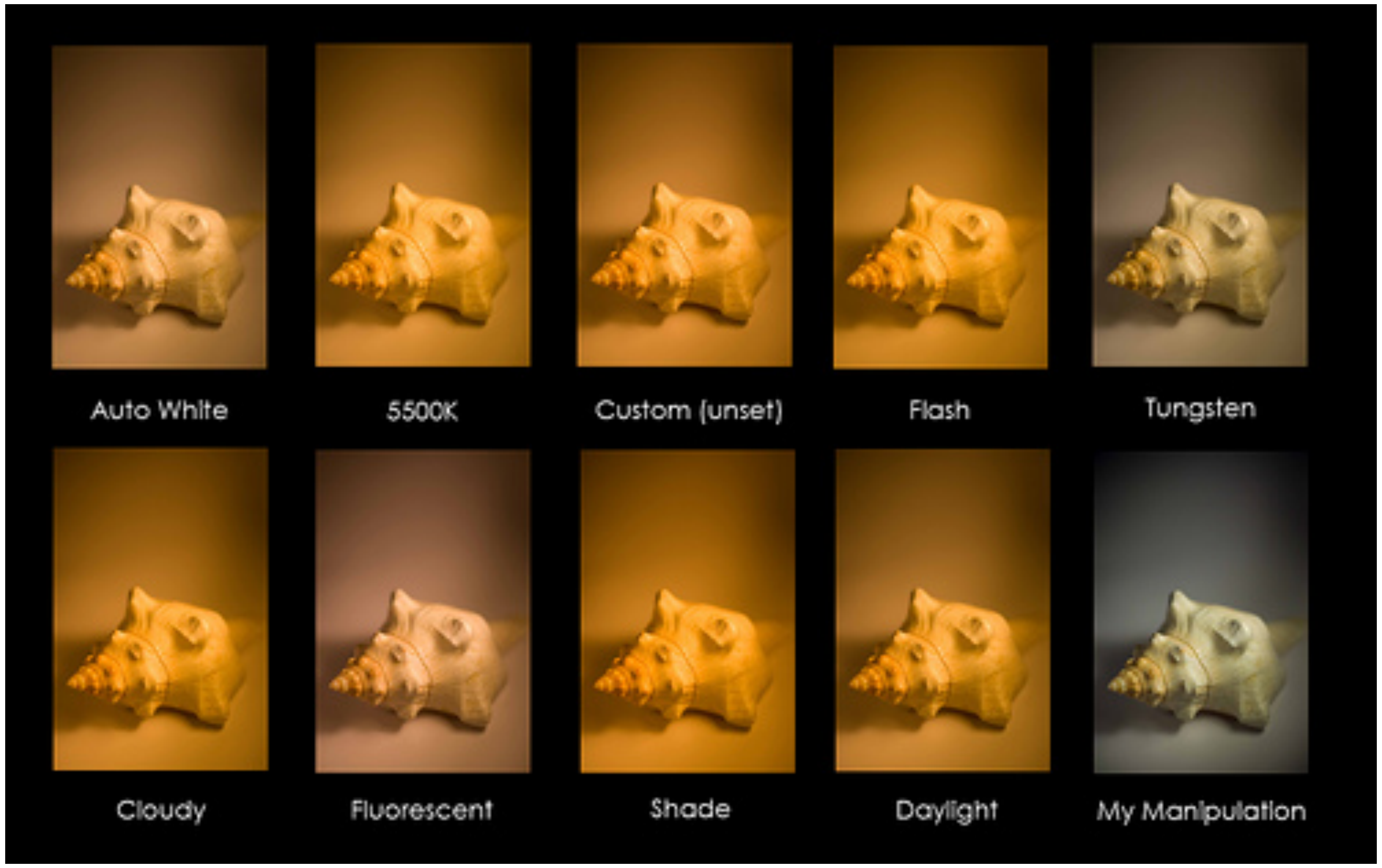
“Dark frame”
(result of exposure with shutter closed)

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)
```

- White balance coefficients are determined based on analysis of image contents:
  - Simple auto-white balance algorithms
    - Gray world assumption: make average of all pixels in image gray
    - Find brightest region of image, make it white ([1,1,1])
- Modern cameras have sophisticated (heuristic-based) white-balance algorithms



Common data-driven solution:

1. Compute features from input image (e.g., histogram)
2. Find similar images in database of images for which good white balance settings are known
3. Use white balance settings from database image

Image credit: basedigitalphotography.com

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

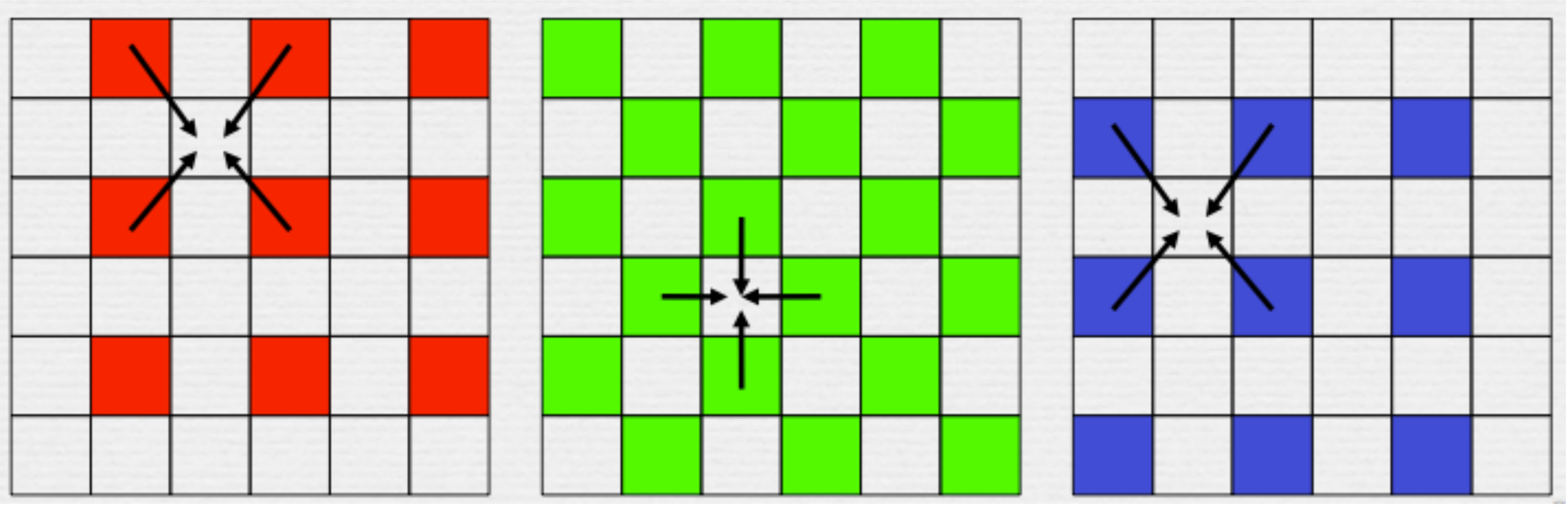
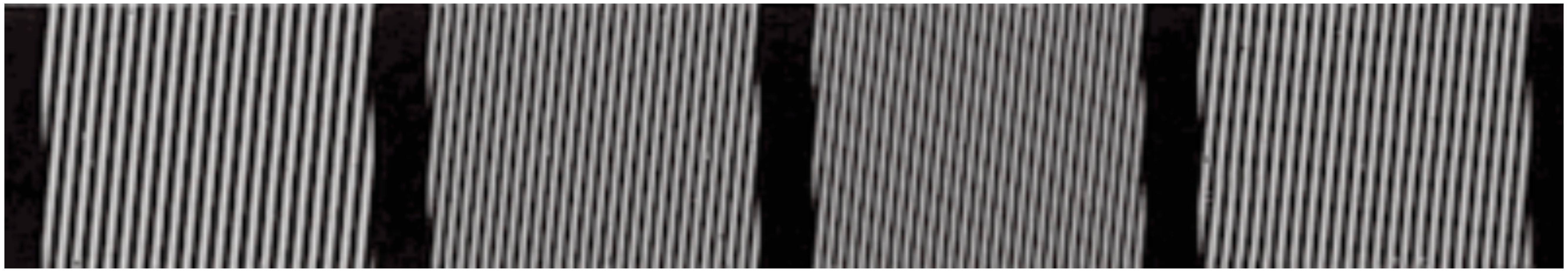


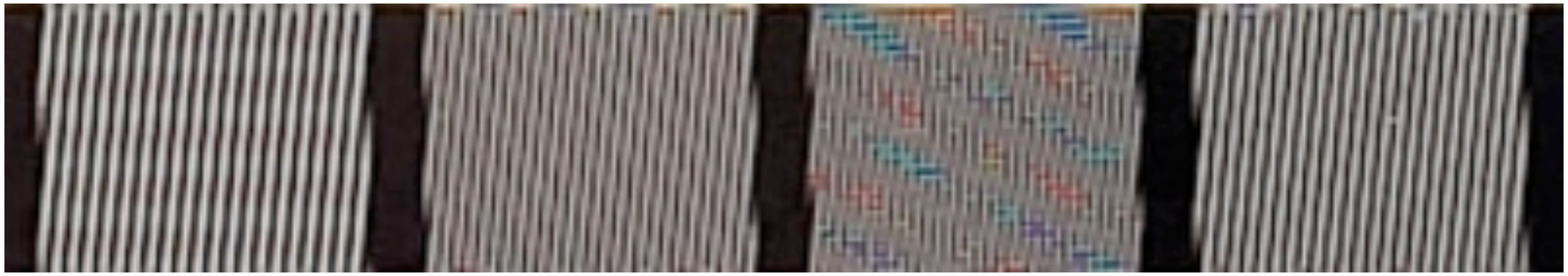
Image credit: Mark Levoy

Demosaicing errors

- **Moire pattern color artifacts**
  - Common difficult case: fine diagonal black and white stripes
  - Common solution:
    - Convert demosaiced value to YCbCr
    - Low-pass filter (blur) CbCr channels
    - Combine filtered CbCr with full resolution Y from sensor to get RGB



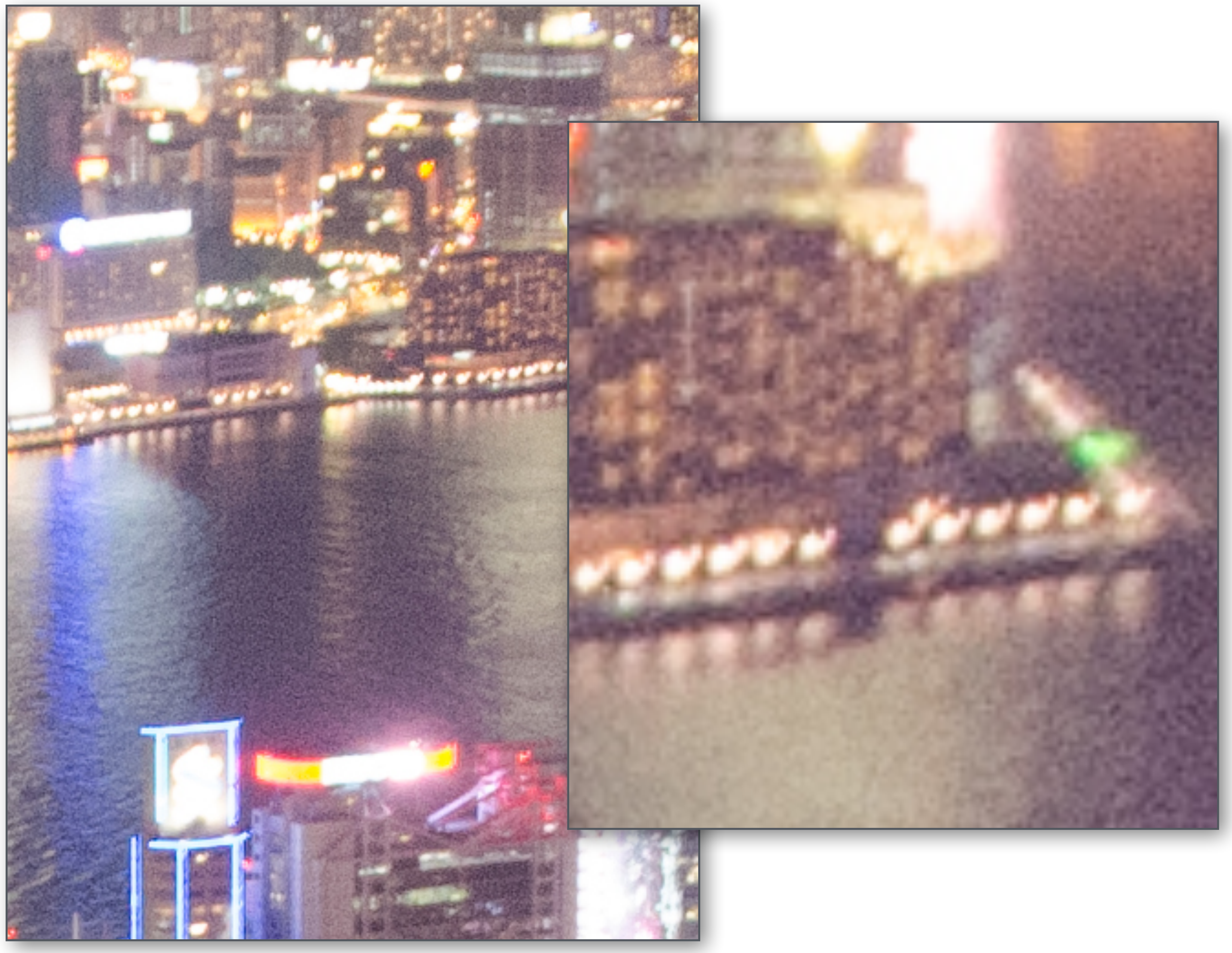
RAW data from sensor

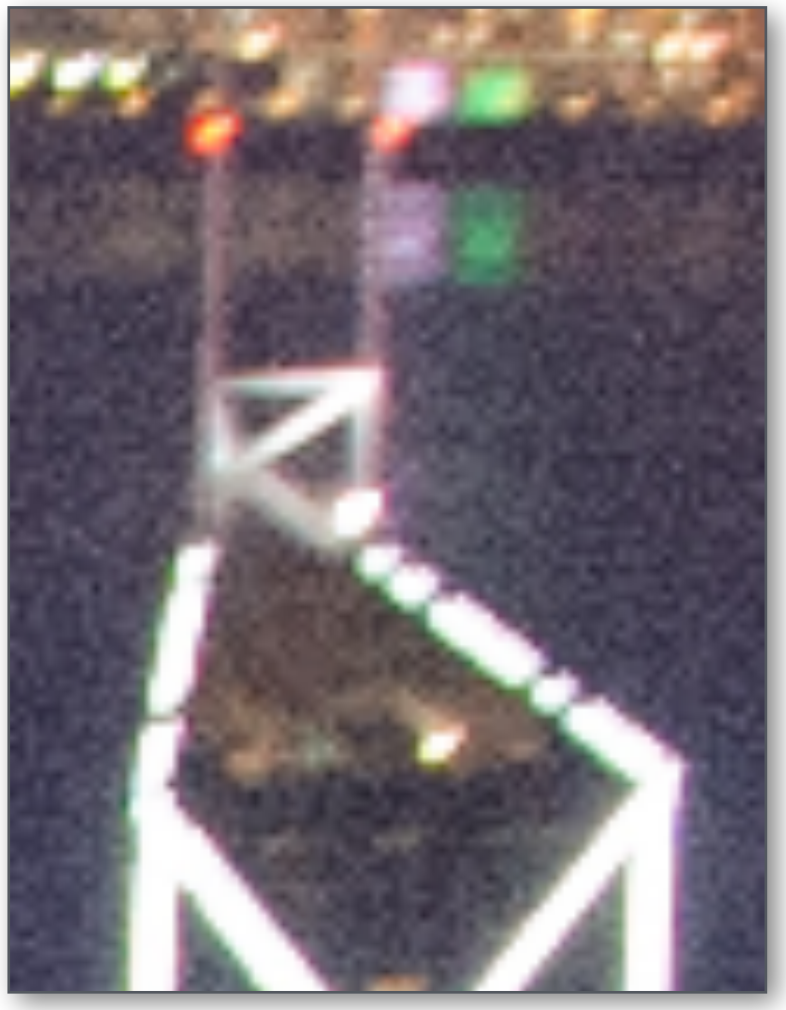


Demosaiced

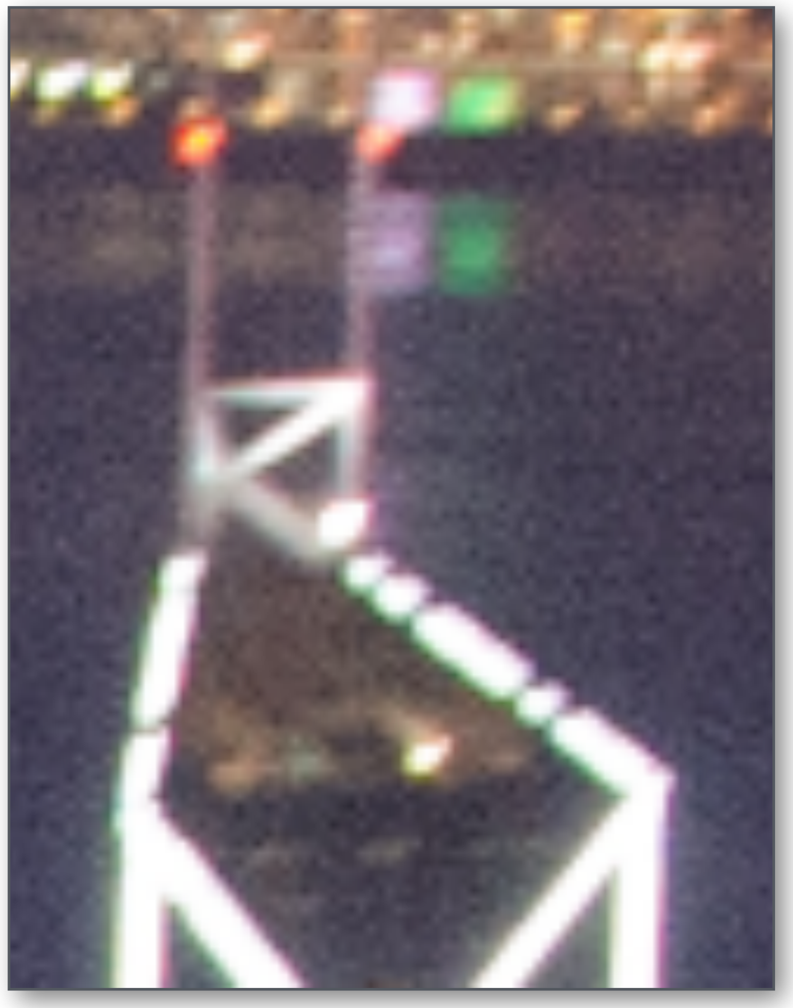
Image credit: http://almanazir.typepad.com/almanazir/2006/11/how_a_camera_ph_1.html

Denoising: effect of downsizing on image noise





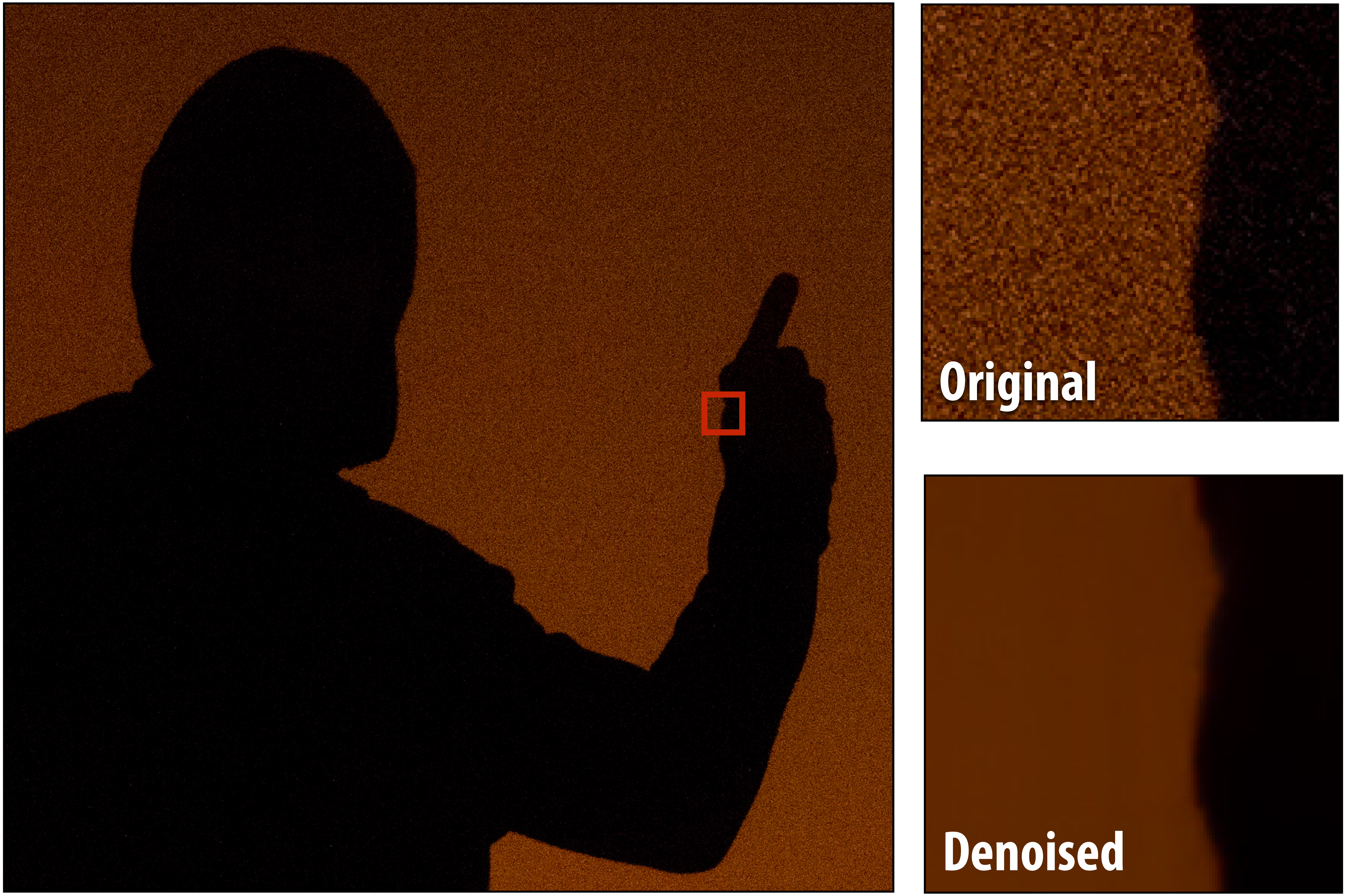
point sampled



averaged down (bilinear resampling)

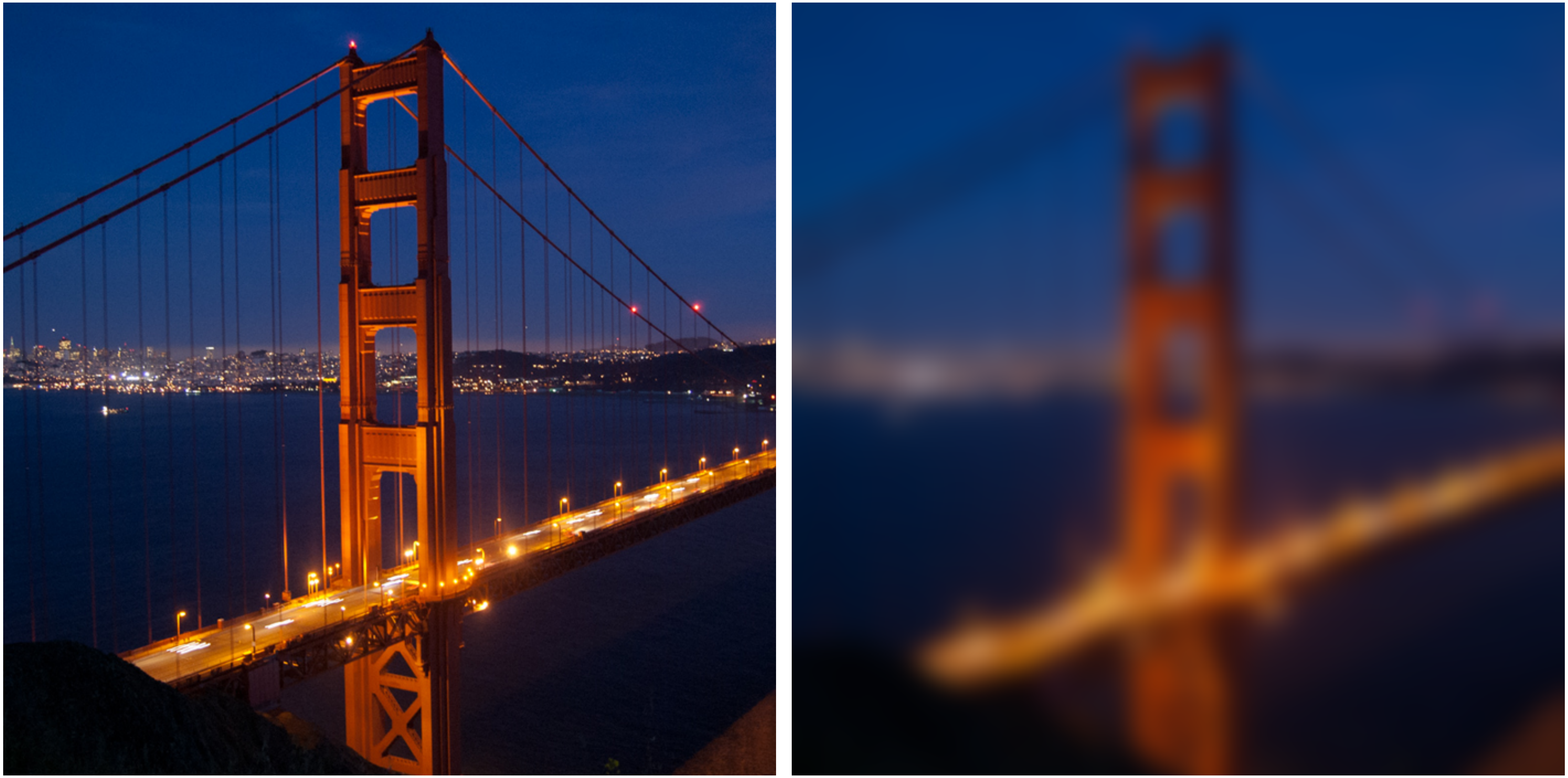
Credit: Levoy

Denoising



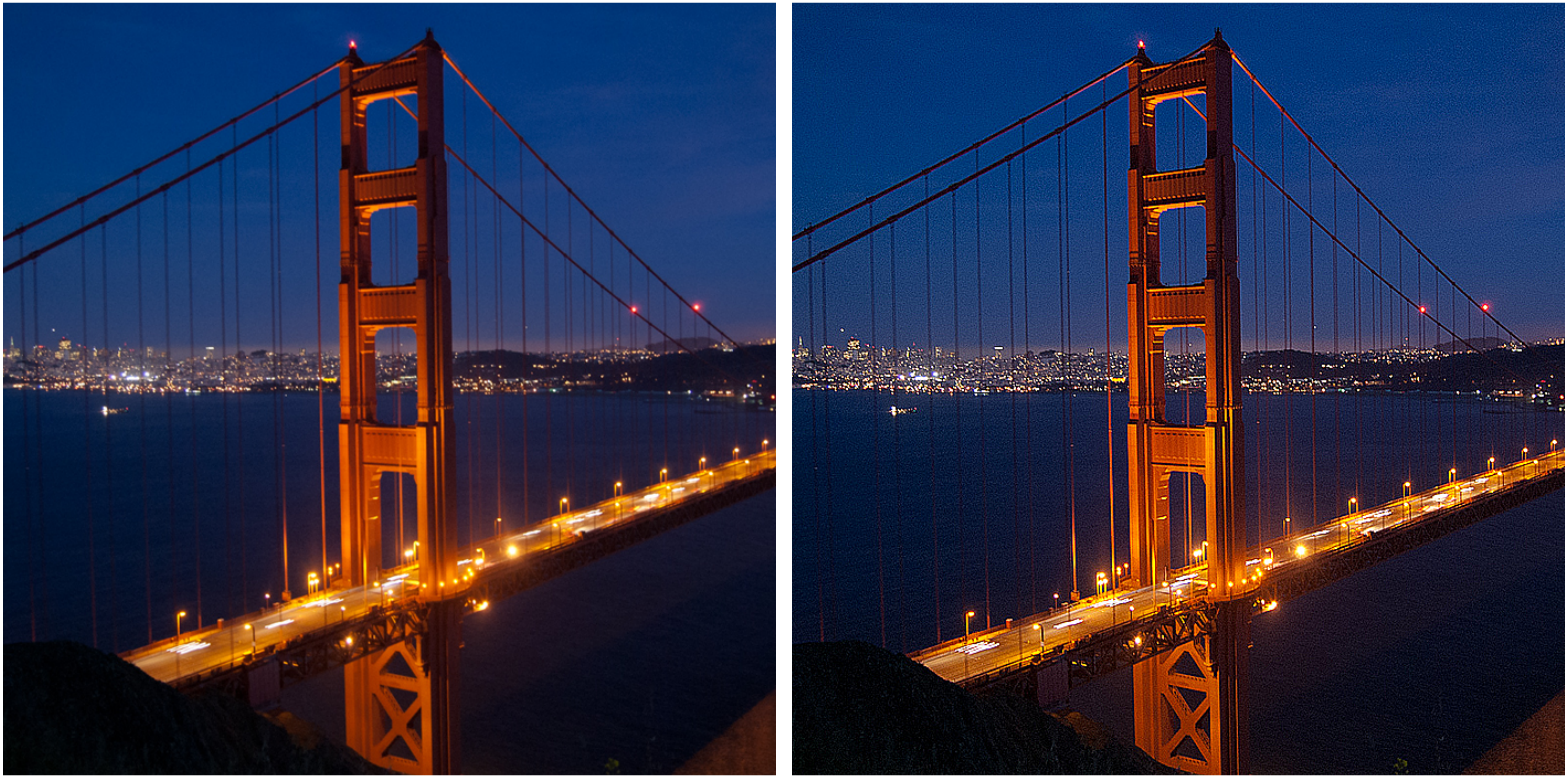
Aside: image processing basics

Example image processing operations



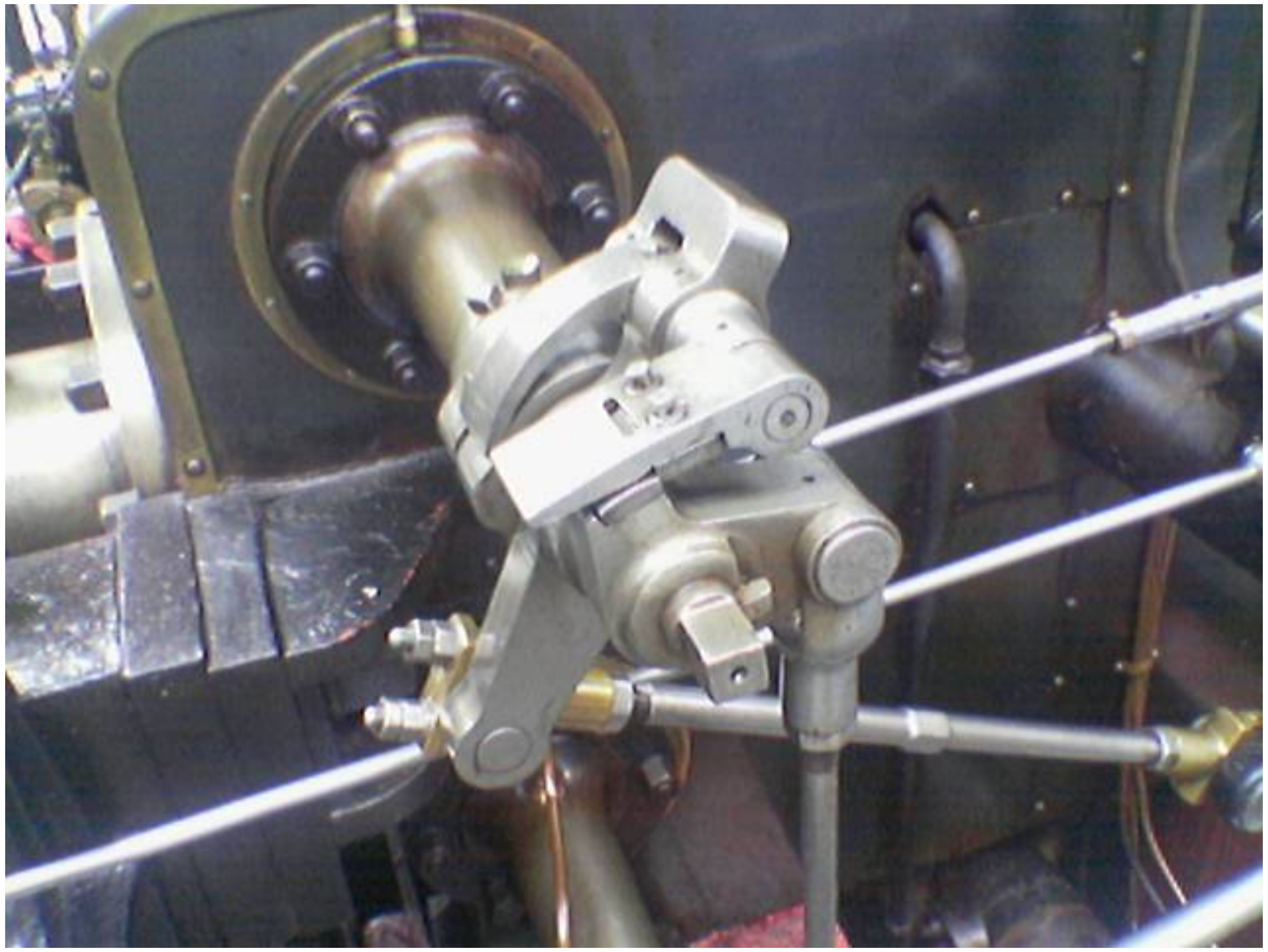
Blur

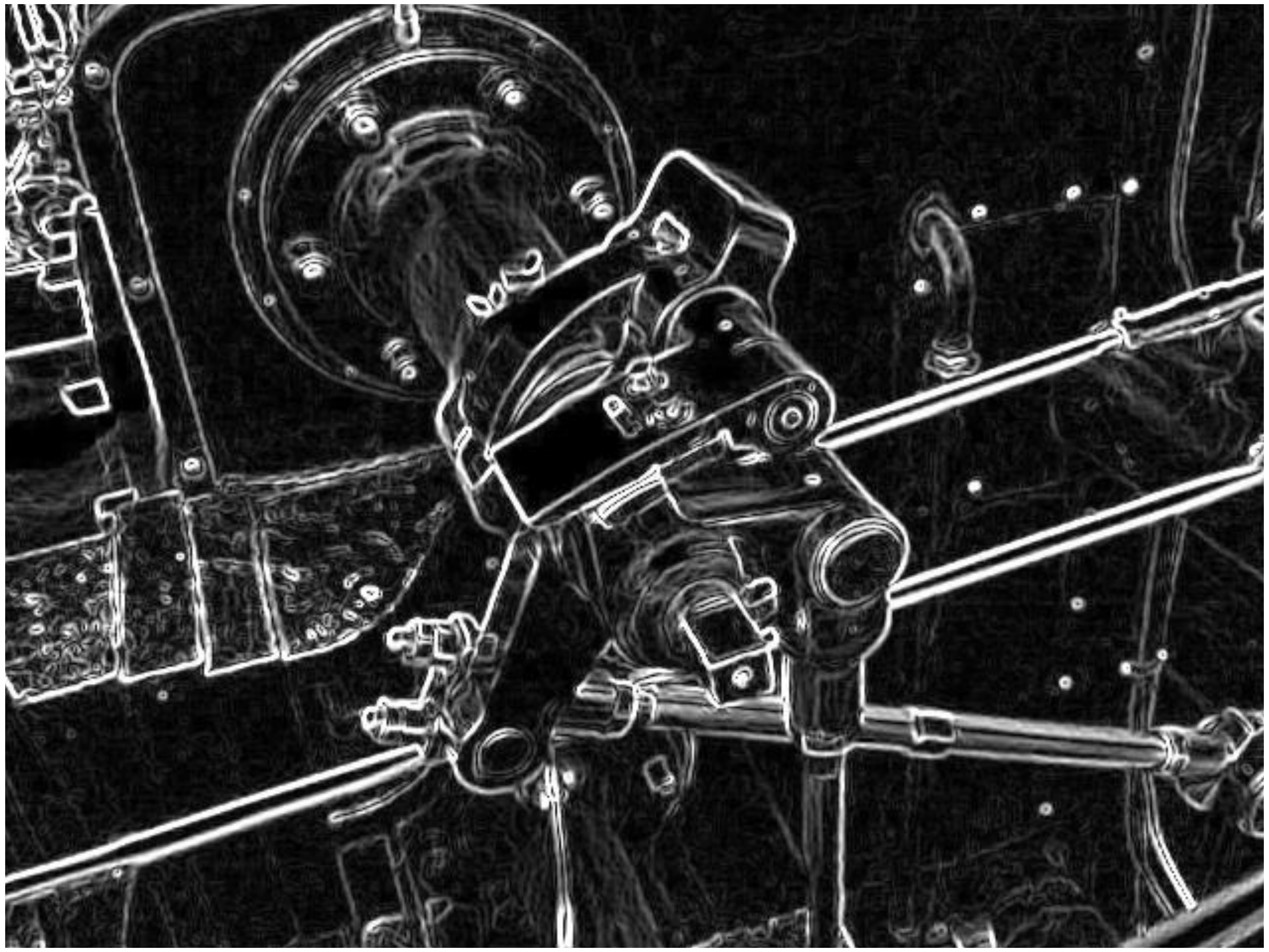
Example image processing operations



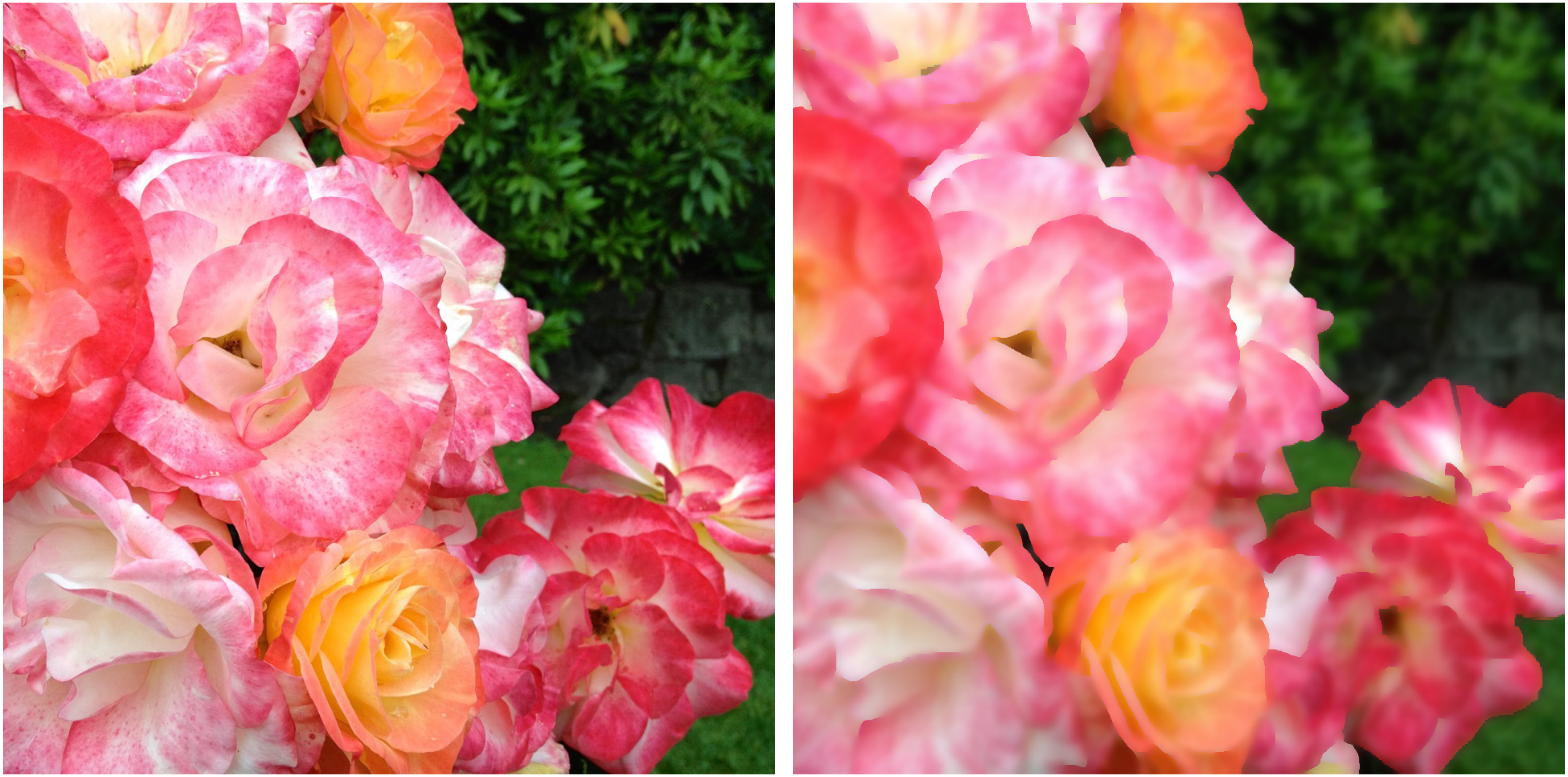
Sharpen

Edge detection





A “smarter” blur (doesn’t blur over edges)

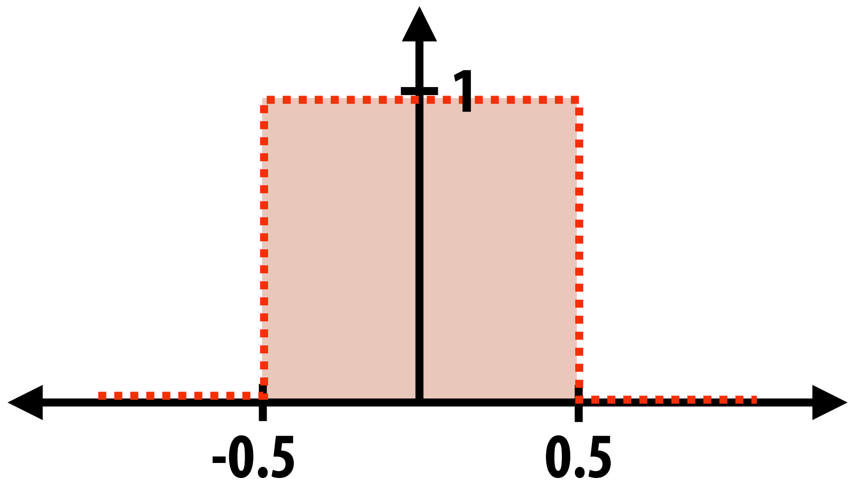


Review: convolution

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

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

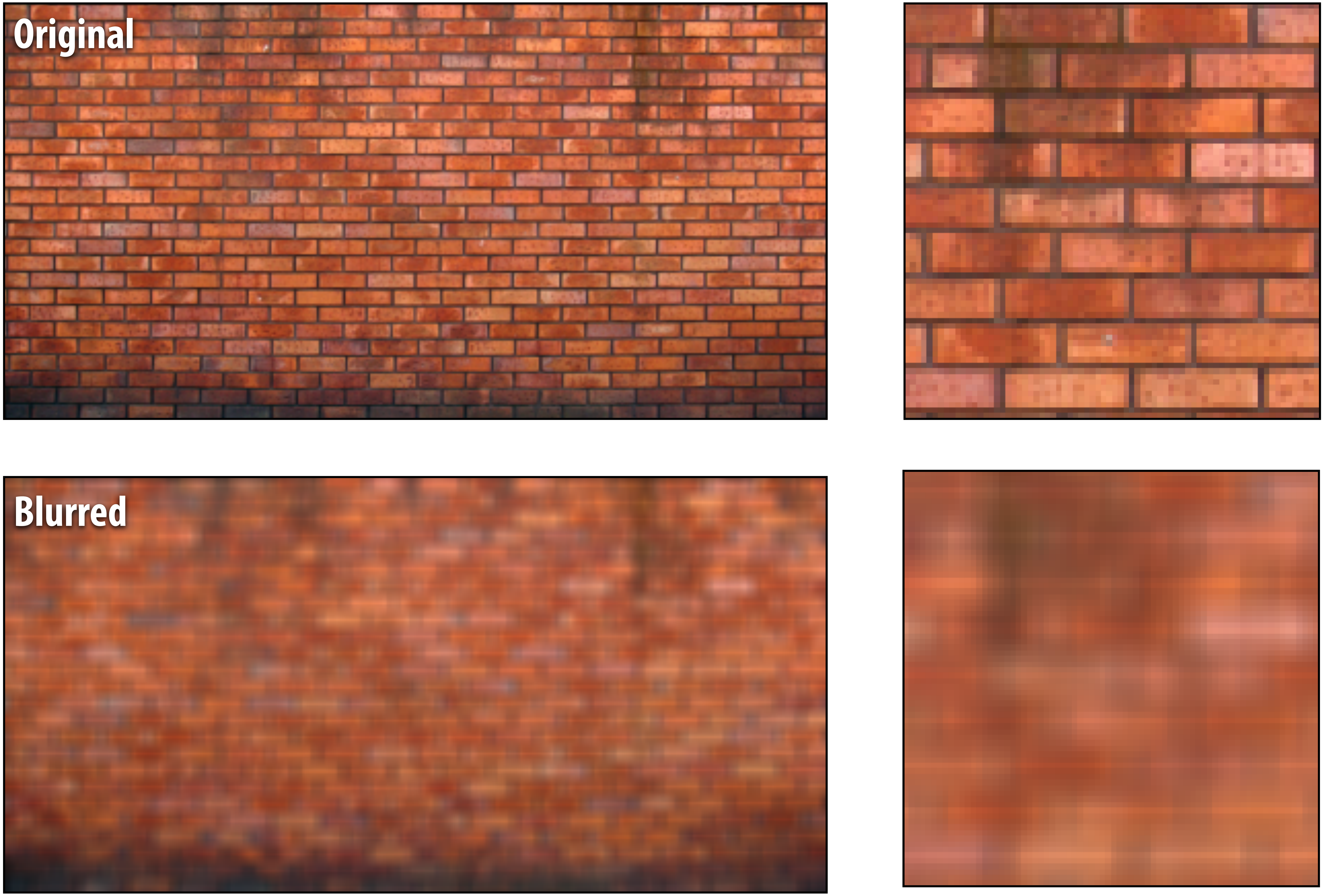
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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

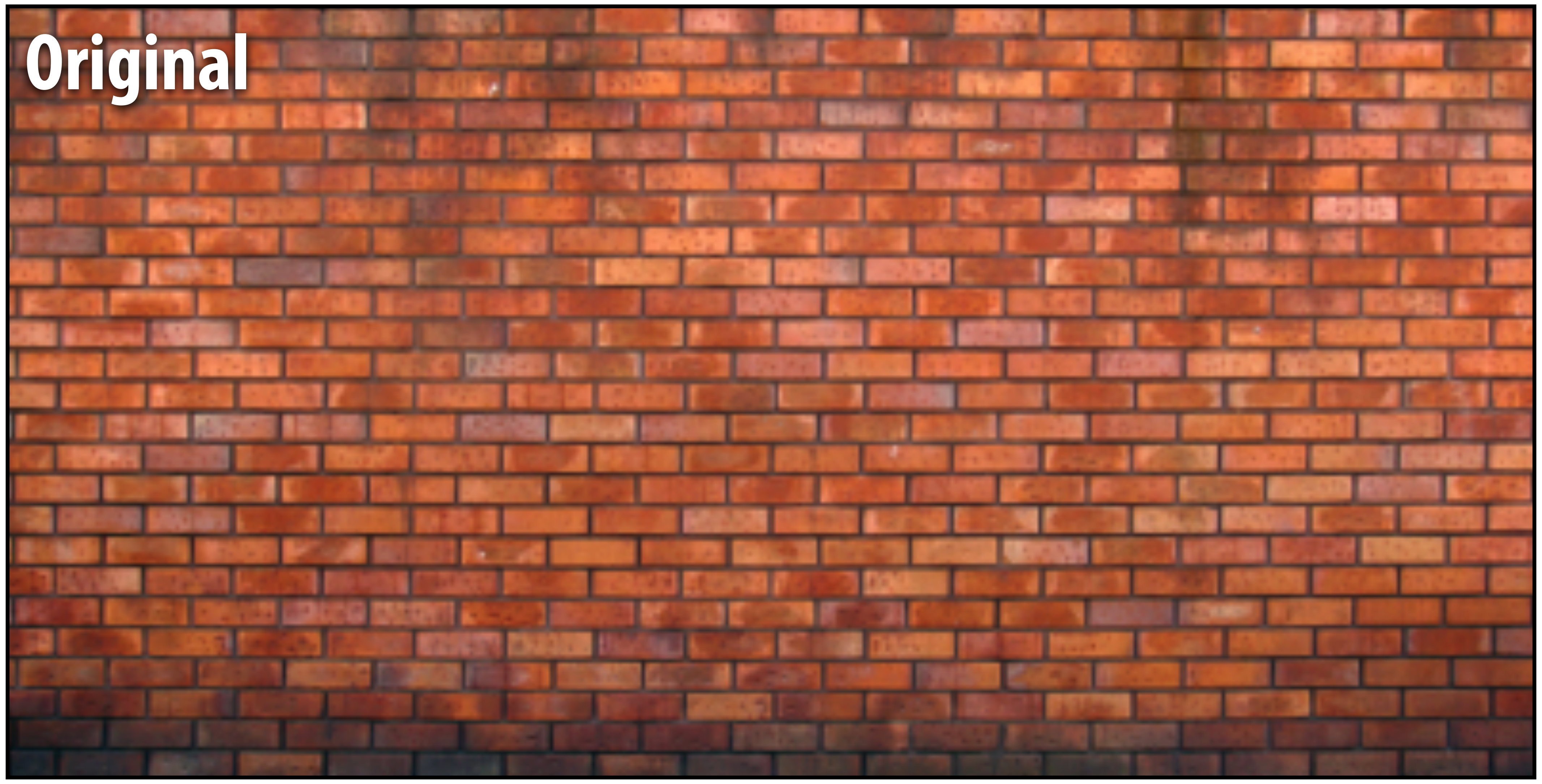
- **Obtain filter coefficients from sampling 2D Gaussian**

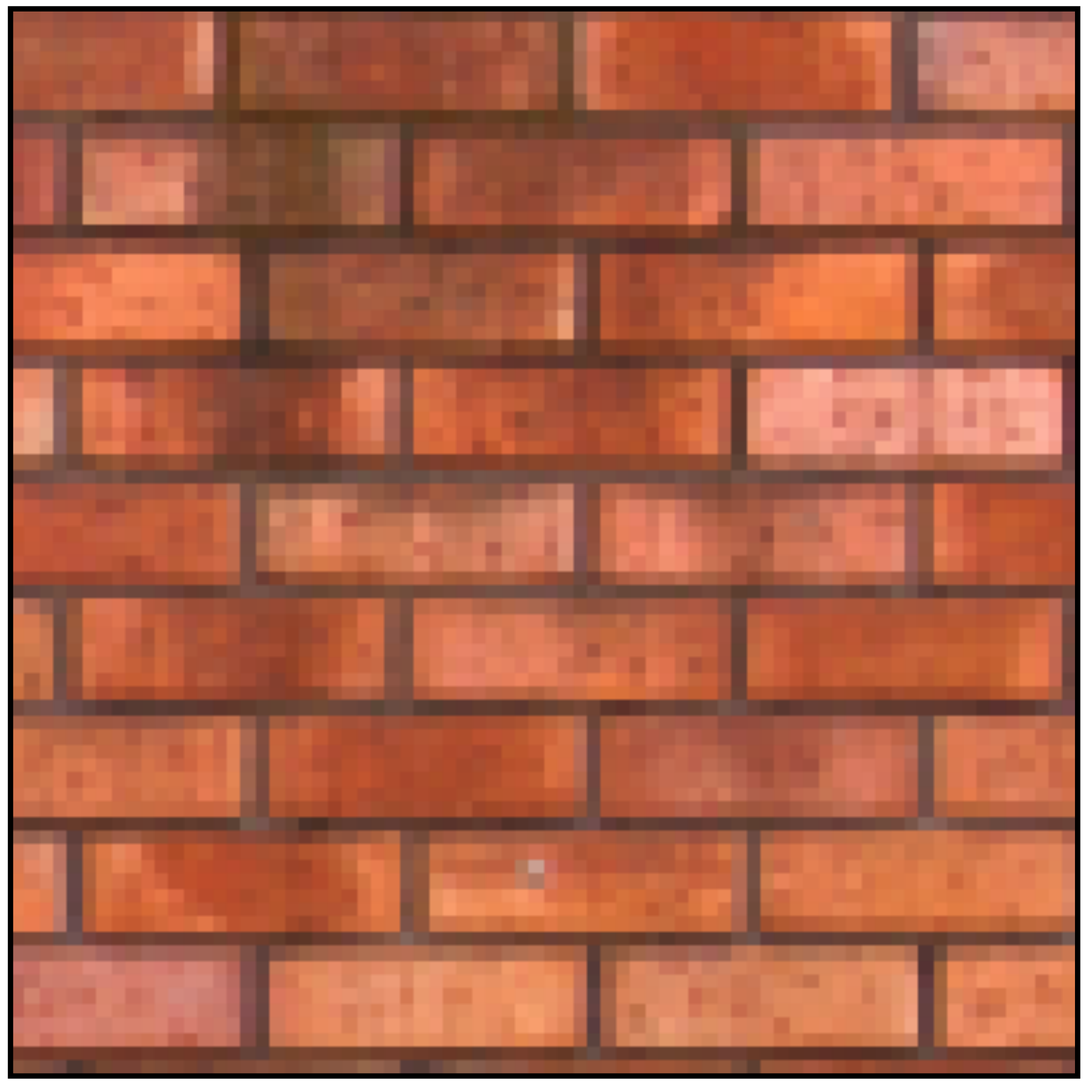
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

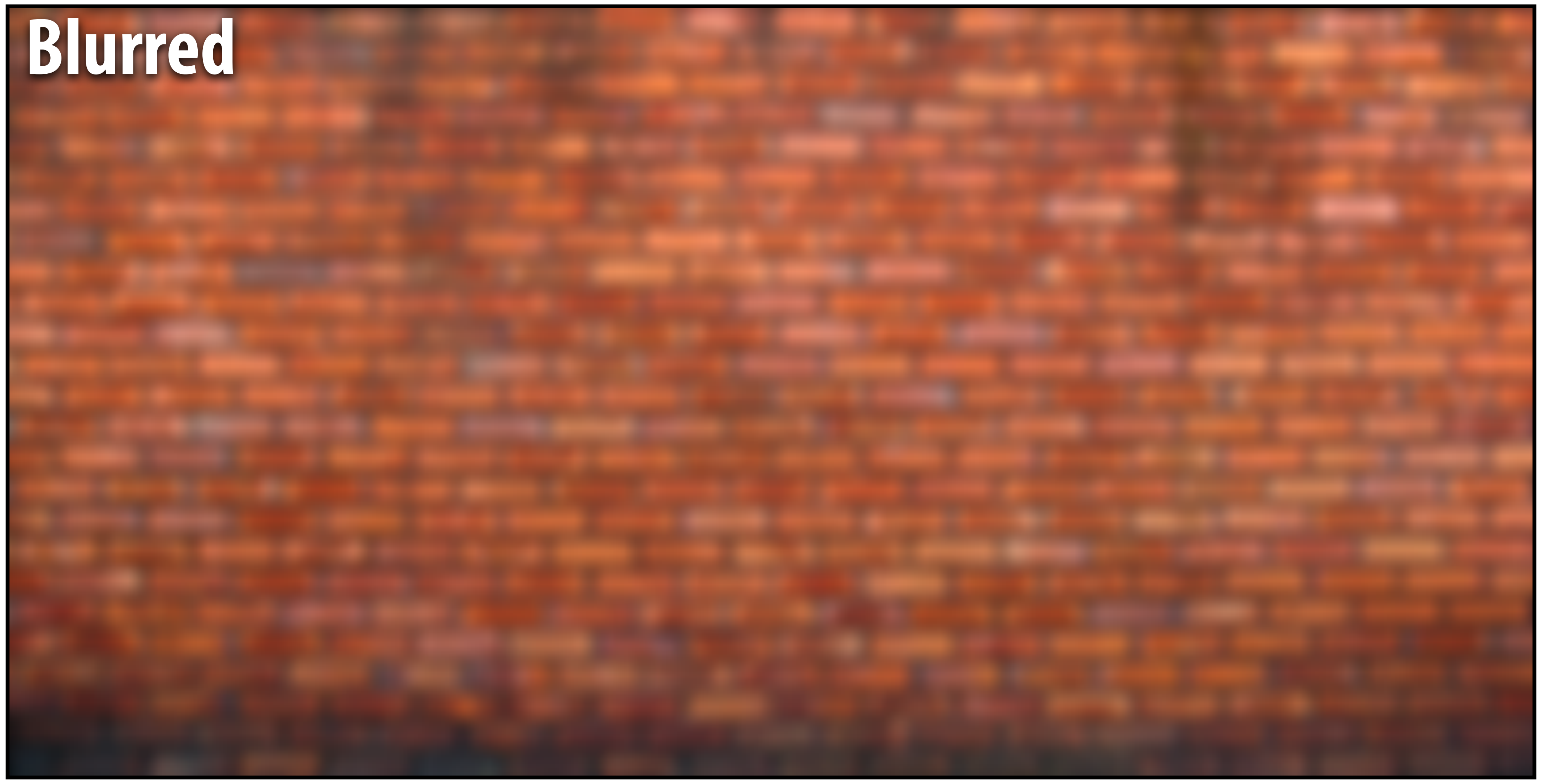
- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

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

7x7 gaussian blur





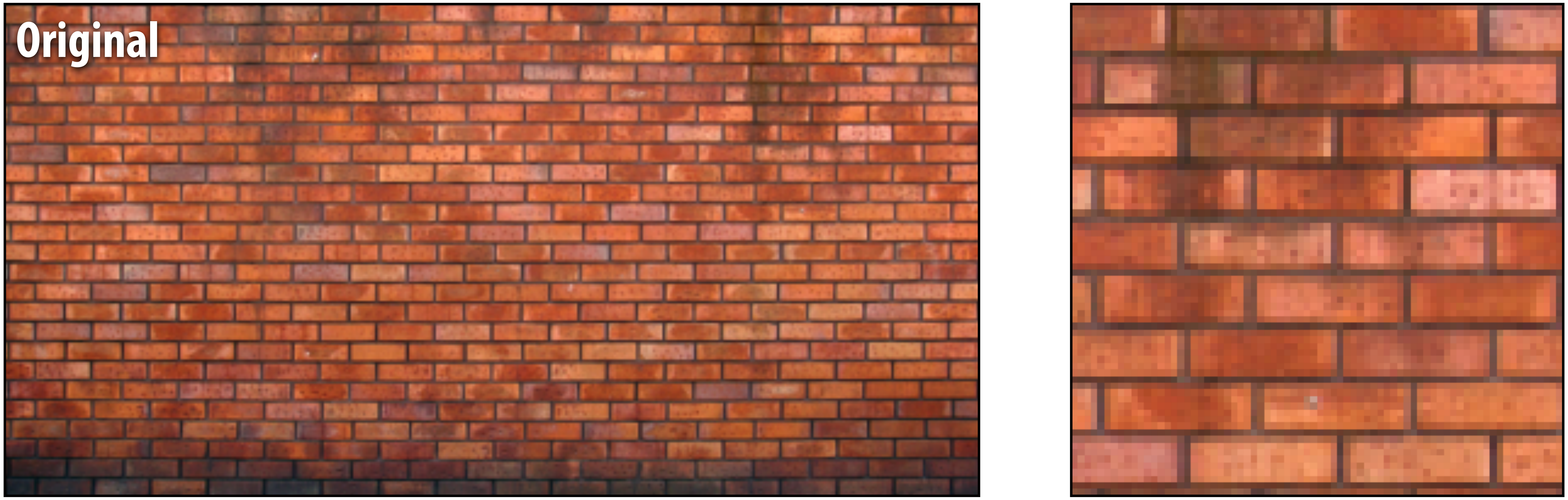


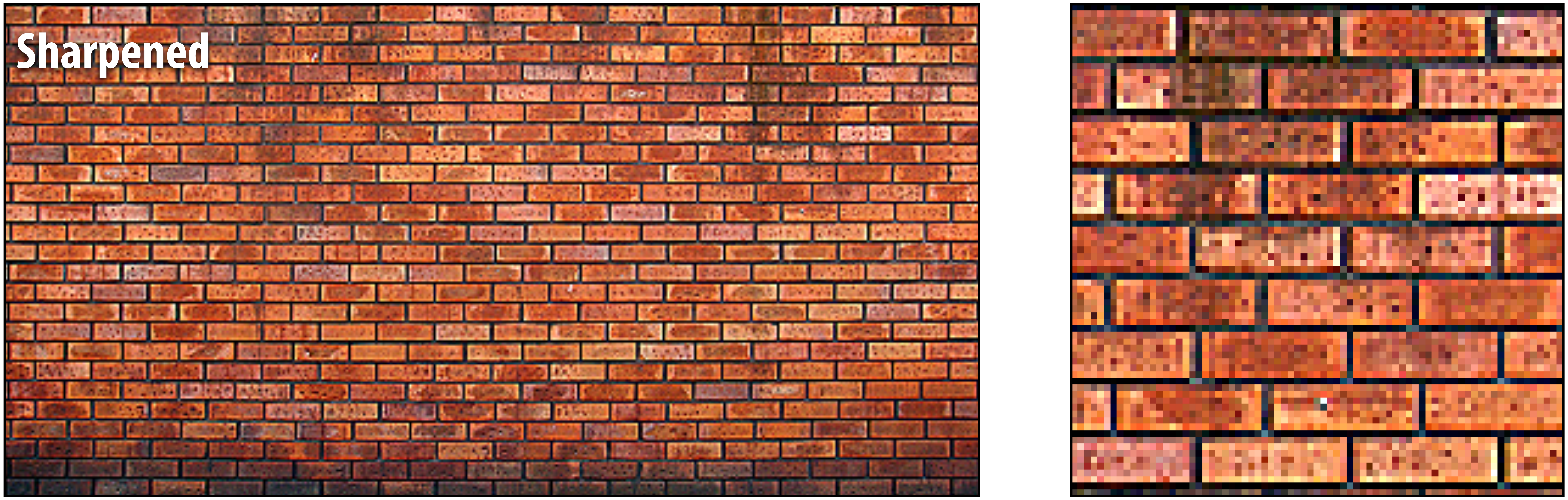
What does convolution with this filter do?

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

Sharpens image!

3x3 sharpen filter





What does convolution with these filters do?

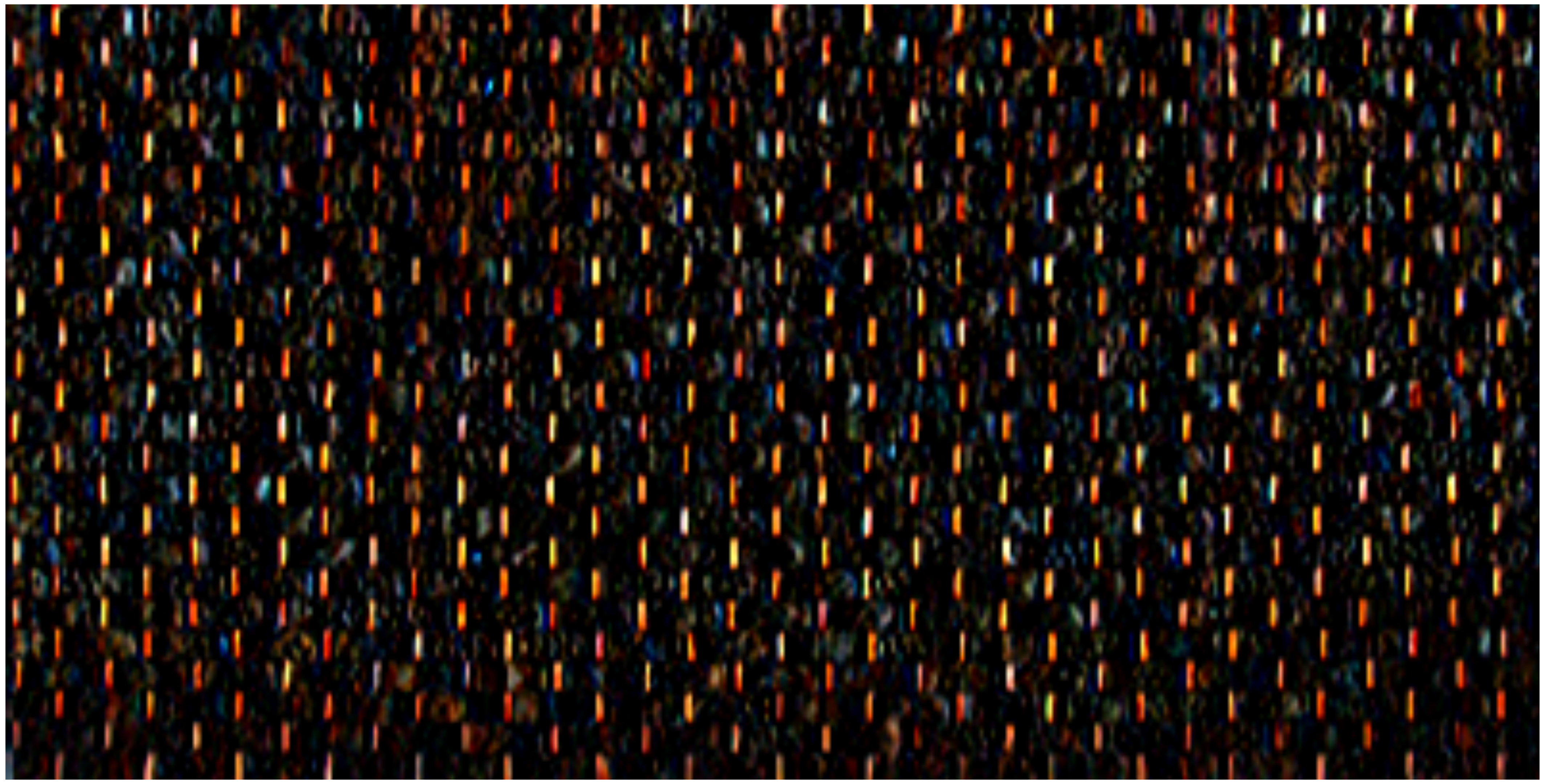
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

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

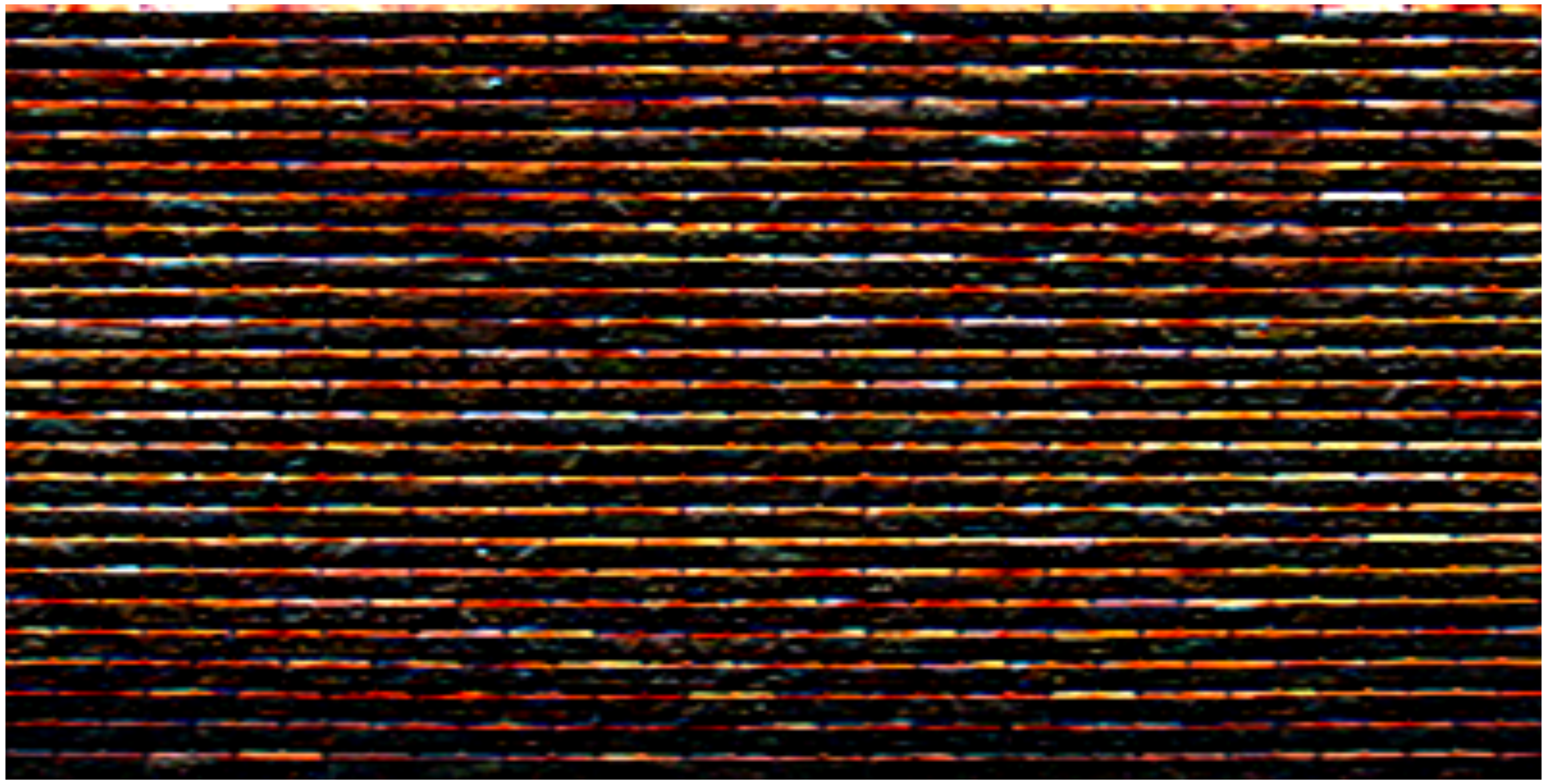
Extracts horizontal gradients

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

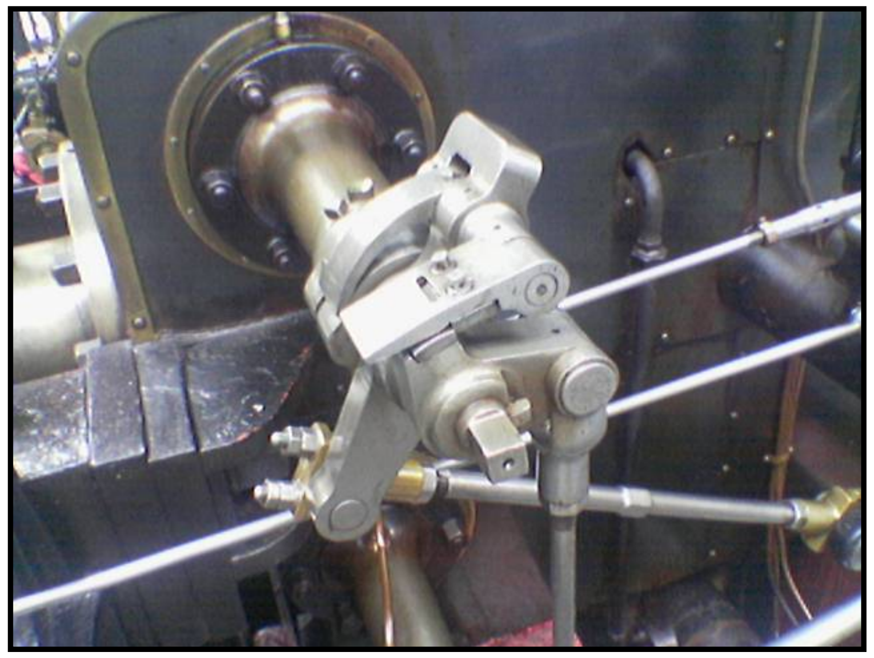
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

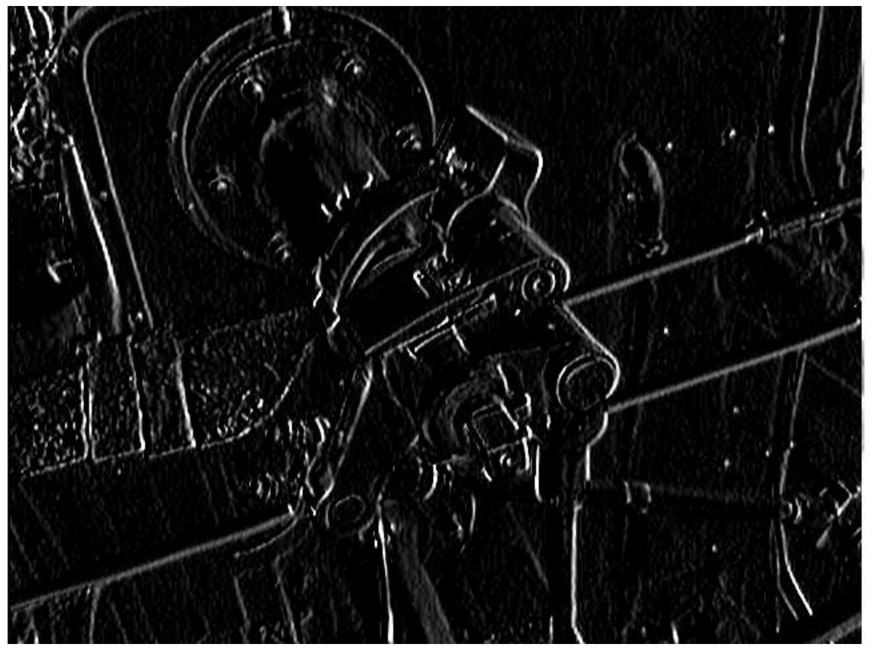
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I$$

- Find pixels with large gradients

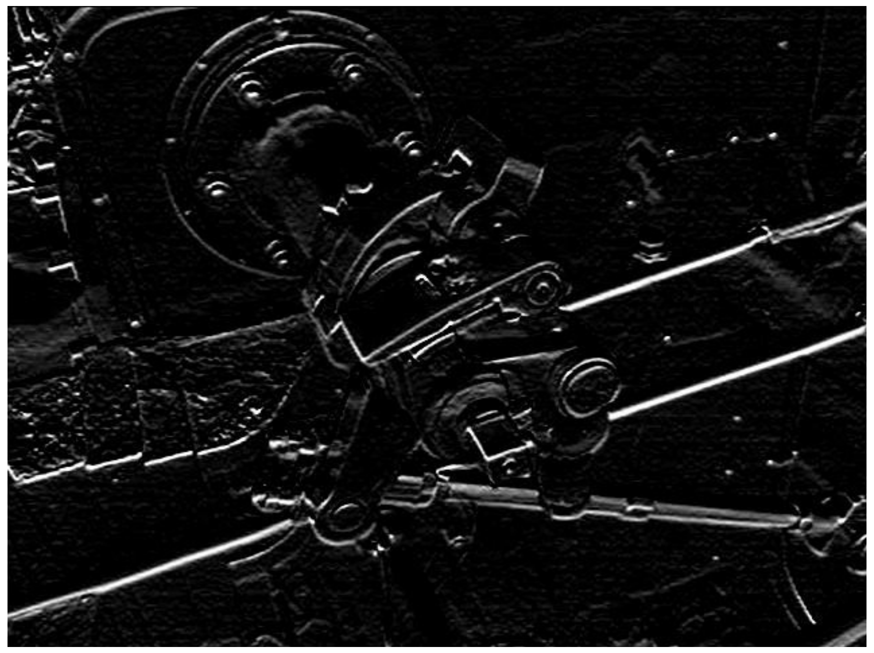
$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

Pixel-wise operation on images

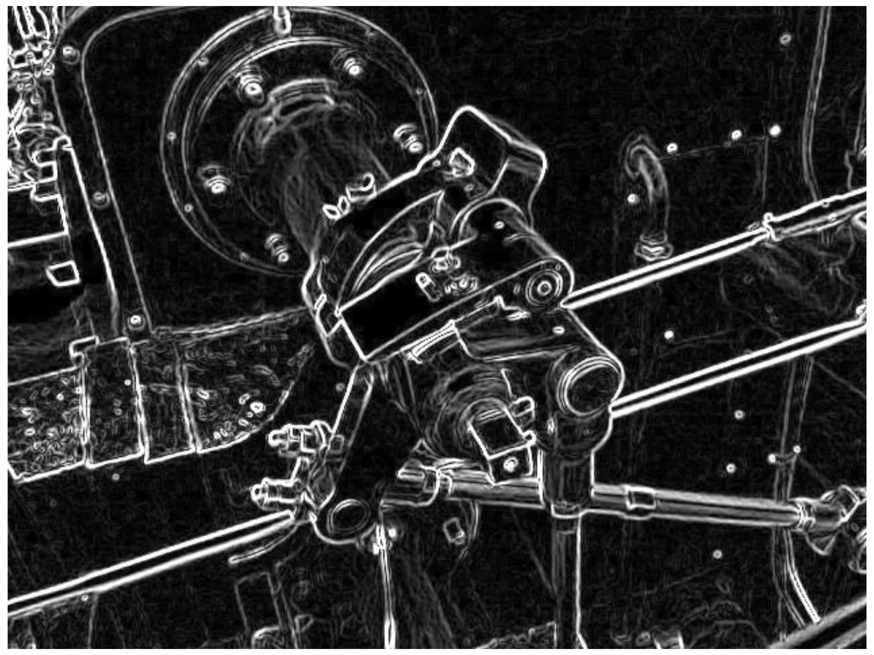




G_x



G_y



G

Cost of convolution with N x N filter?

```
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;
  }
}
```

In this 3x3 box blur example:

Total work per image = 9 x WIDTH x HEIGHT

For N x N filter: N^2 x WIDTH x HEIGHT

Separable filter

- A filter is separable if can bee written as the outer product of two other filters. Example: a 2D box blur

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \frac{1}{3}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix}$$

  - Exercise: write 2D gaussian and vertical/horizontal gradient detection filters as product of 1D filters (they are separable!)

- Key property: 2D convolution with separable filter can be written as two 1D convolutions!

Implementation of 2D box blur via two 1D convolutions

```
int WIDTH = 1024
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1./3, 1./3, 1./3};

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

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++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

Total work per image for NxN filter:

2N x WIDTH x HEIGHT

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]);
    }
}
```

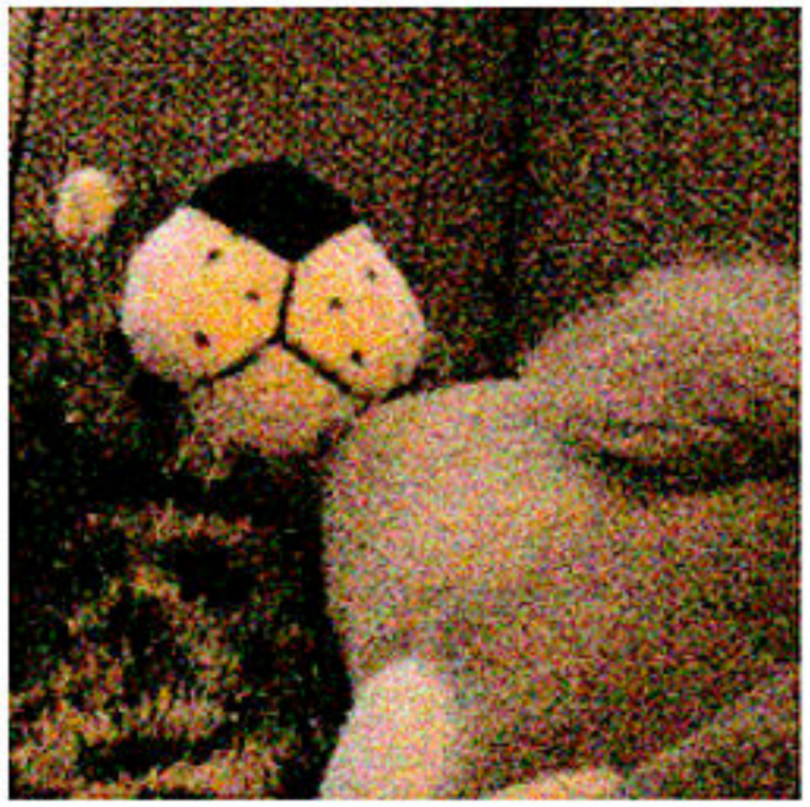
This filter clamps pixels to the min/max of its cardinal neighbors (e.g., hot-pixel suppression)

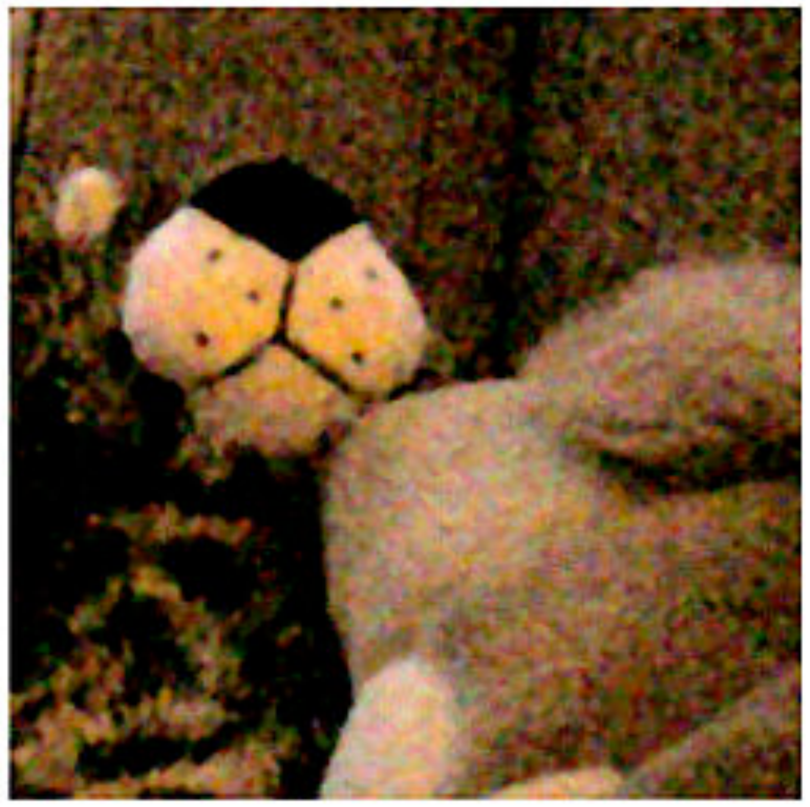
Median filter

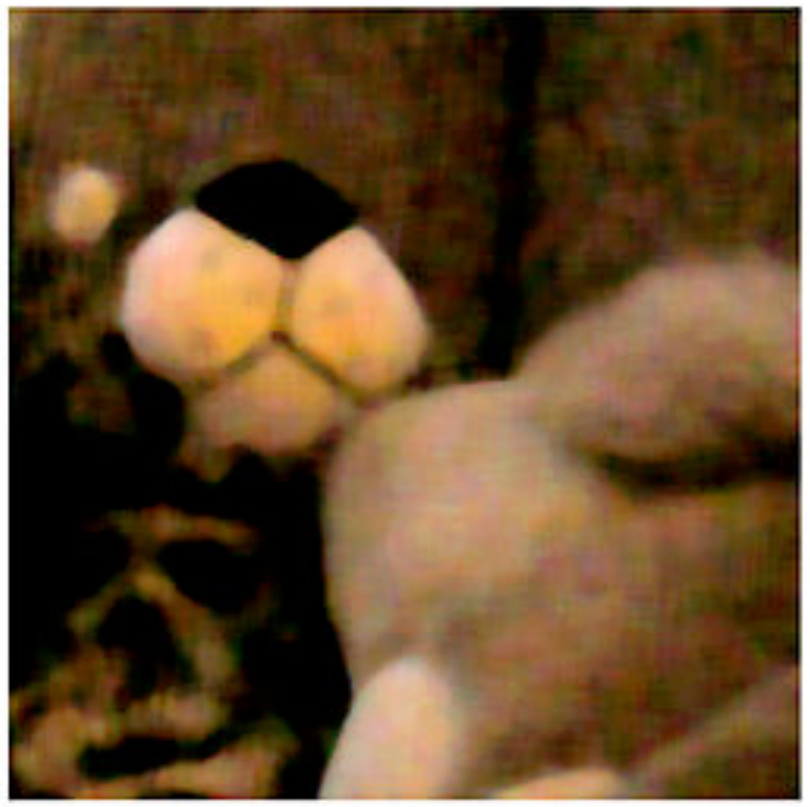
- Replace pixel with median of its neighbors
  - Useful noise reduction filter: unlike gaussian blur, one bright pixel doesn't drag up the average for entire region

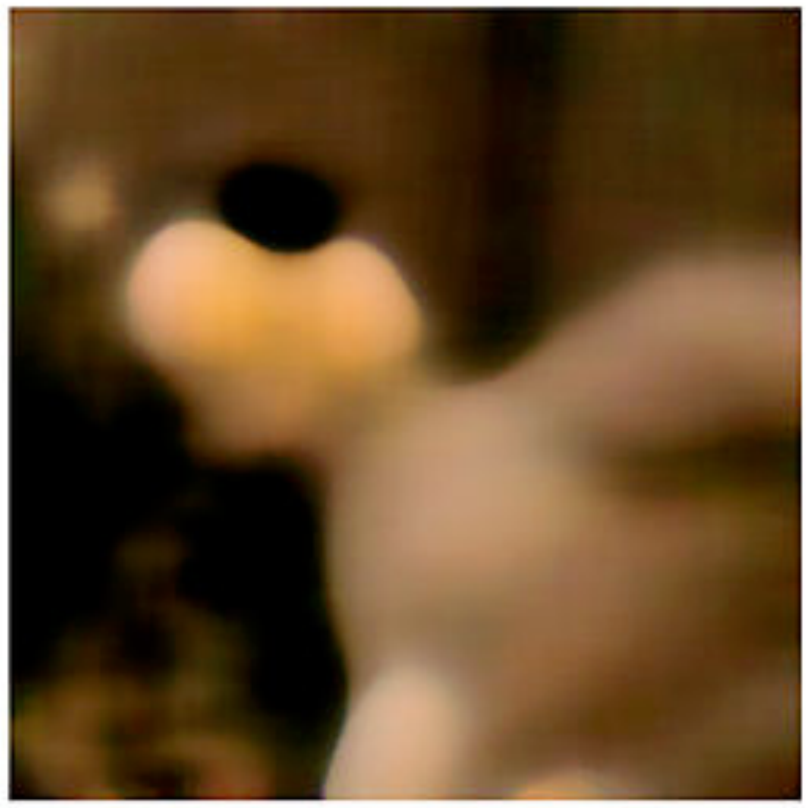
- Not linear, not separable
  - Filter weights are 1 or 0 (depending on image content)

```
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

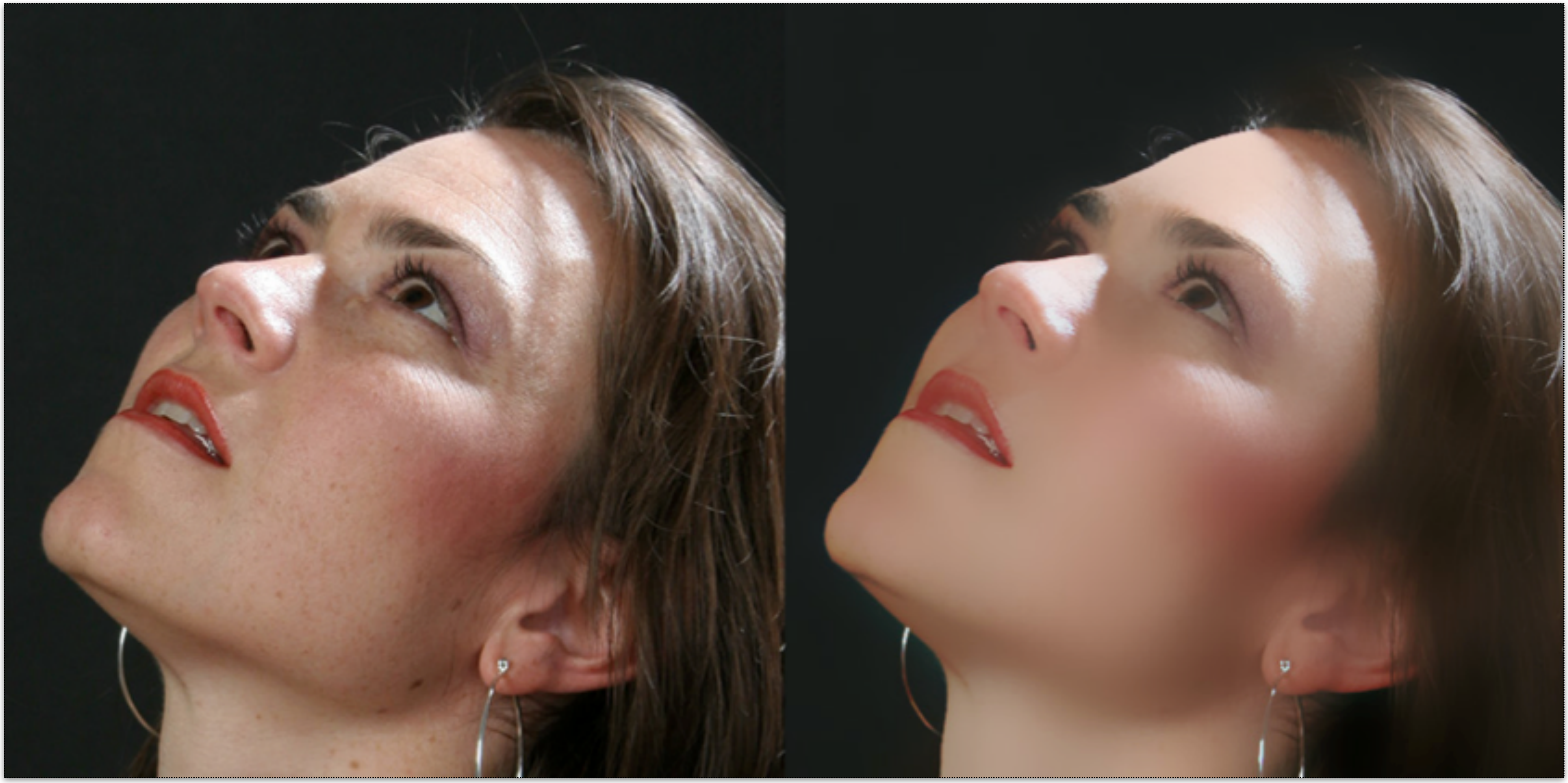
```
int WIDTH = 1024;
int HEIGHT = 1024;
uint8 input[(WIDTH+2) * (HEIGHT+2)];
uint8 output[WIDTH * HEIGHT];
int histogram[256];

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

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

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

- 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: $f(x) = 0 \text{ if } x > \mathit{threshold}, 1 \ \mathit{otherwise}$

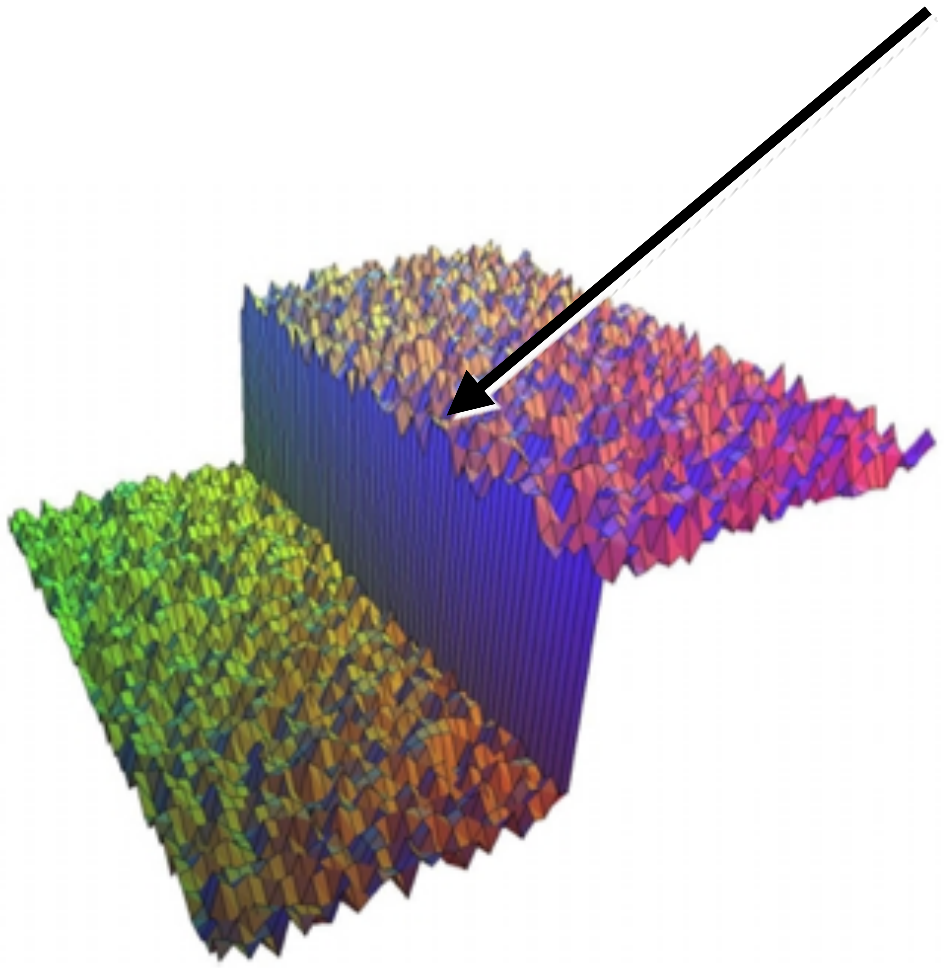
Value of output pixel (x,y) is the weighted sum of all pixels in the support region of a truncated gaussian kernel

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

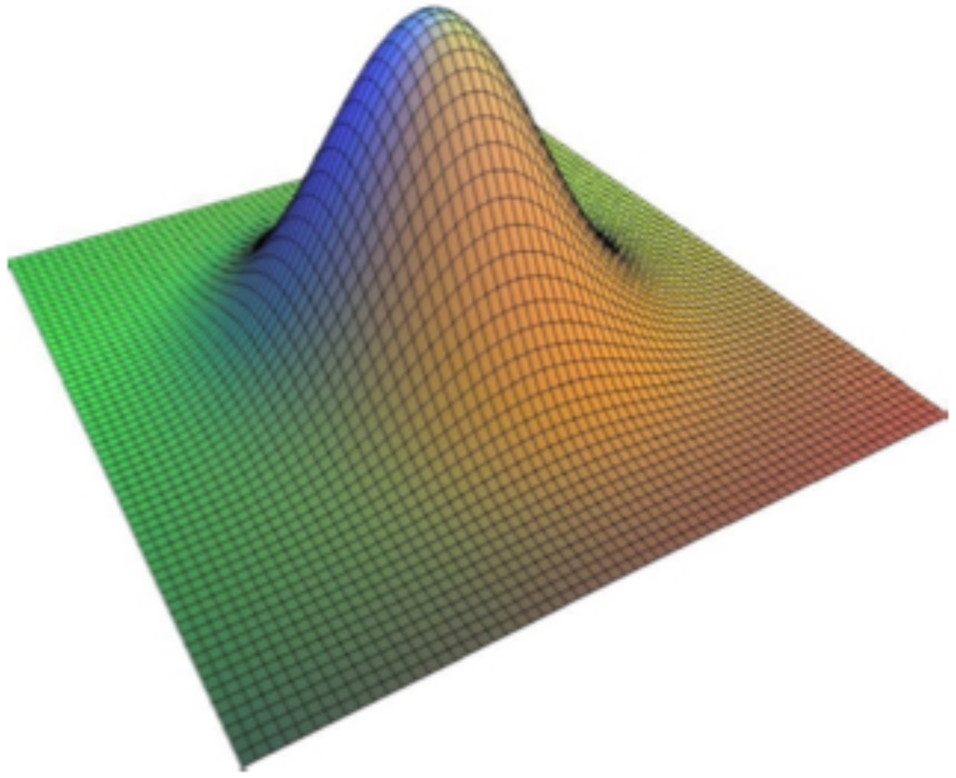
Bilateral filter

Input pixel *p*

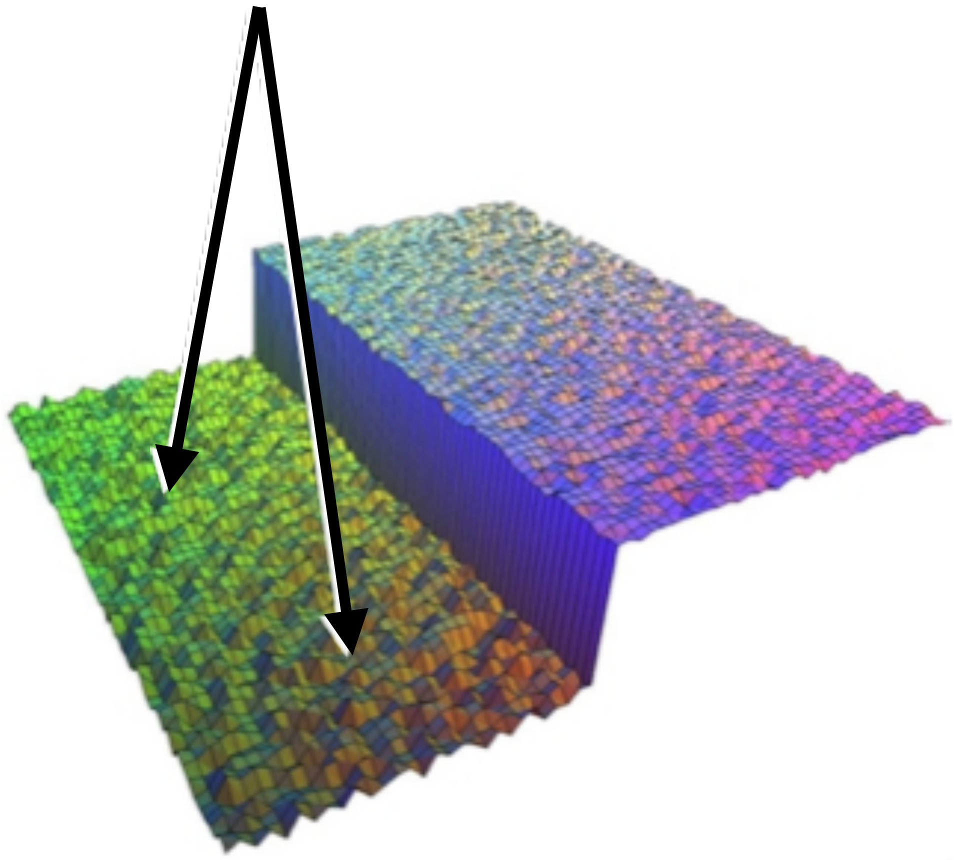
Pixels with significantly different intensity as *p* contribute little to filtered result (they are “on the “other side of the edge”



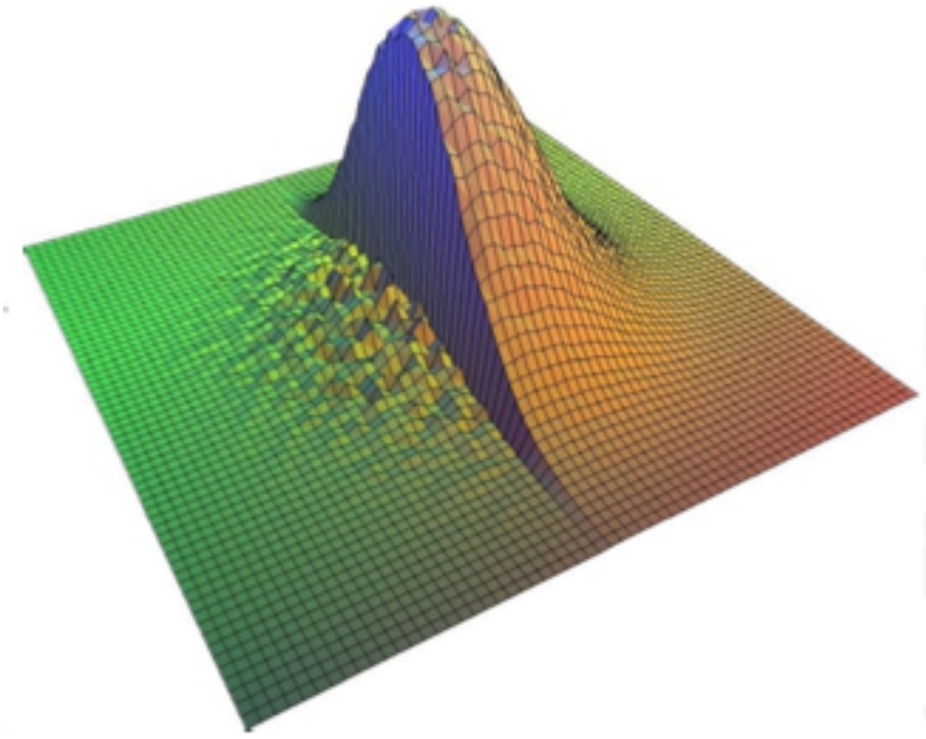
Input image



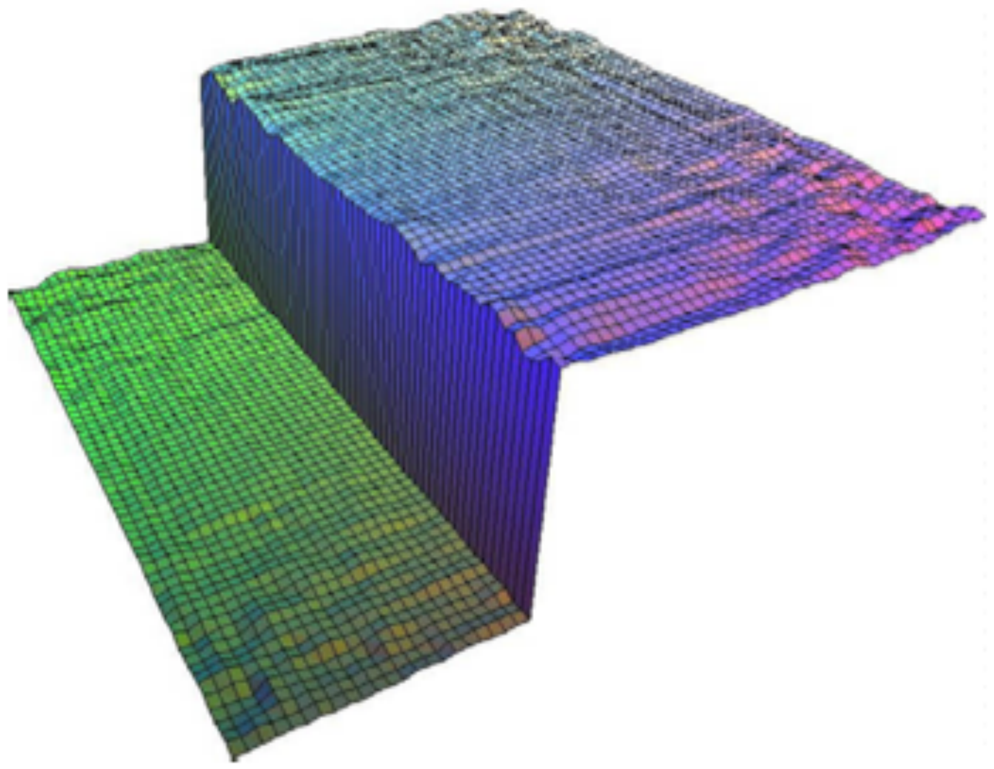
G(): gaussian about input pixel *p*



f(): Influence of support region



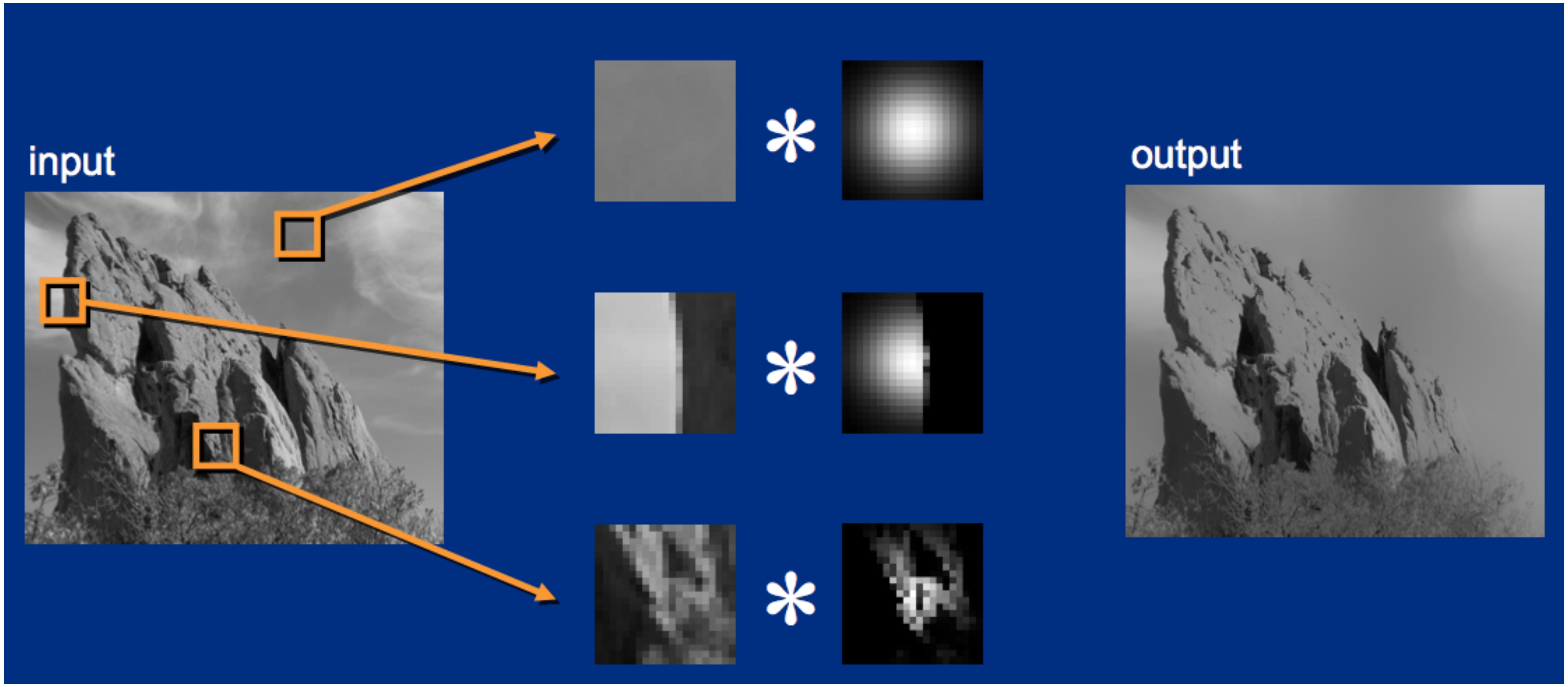
G x f: filter weights for pixel *p*



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)

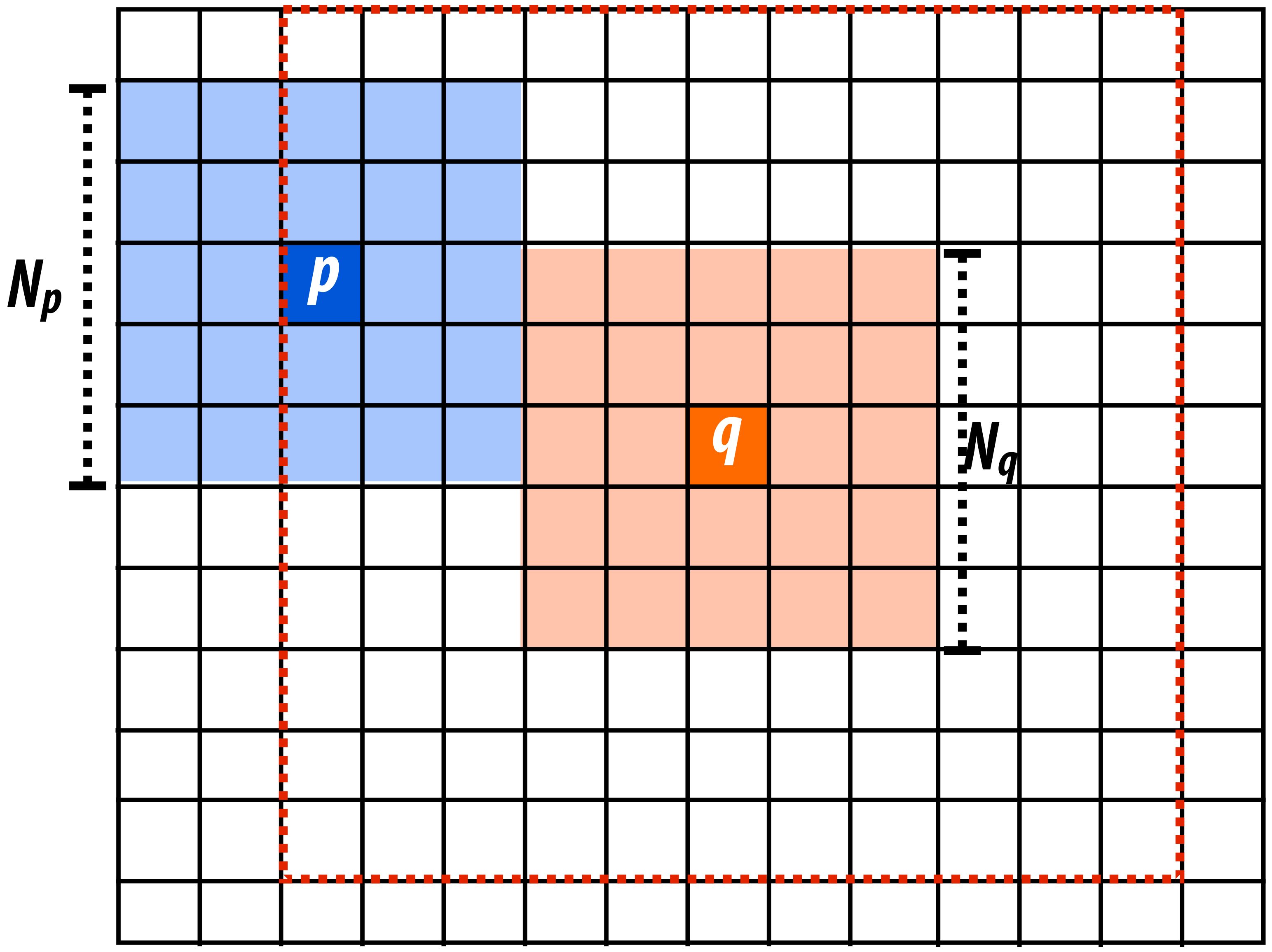
Figure credit: SIGGRAPH 2008 Course: "A Gentle Introduction to Bilateral Filtering and its Applications" Paris et al.

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

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

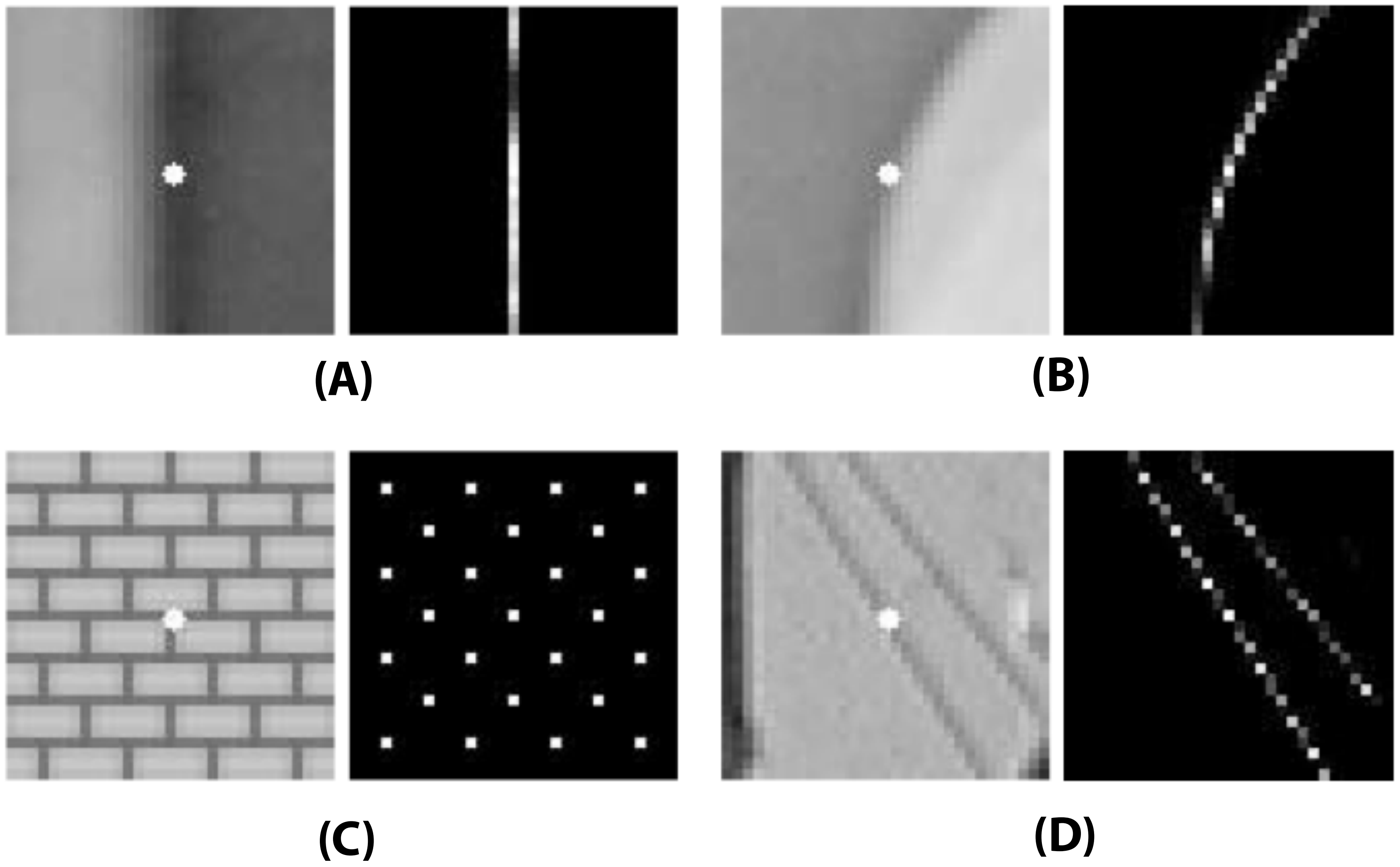
$$w(p,q) = \frac{1}{C_p} e^{\frac{-\| N_p - N_q \|^2}{h^2}}$$



  - 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)
  - Cp 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)

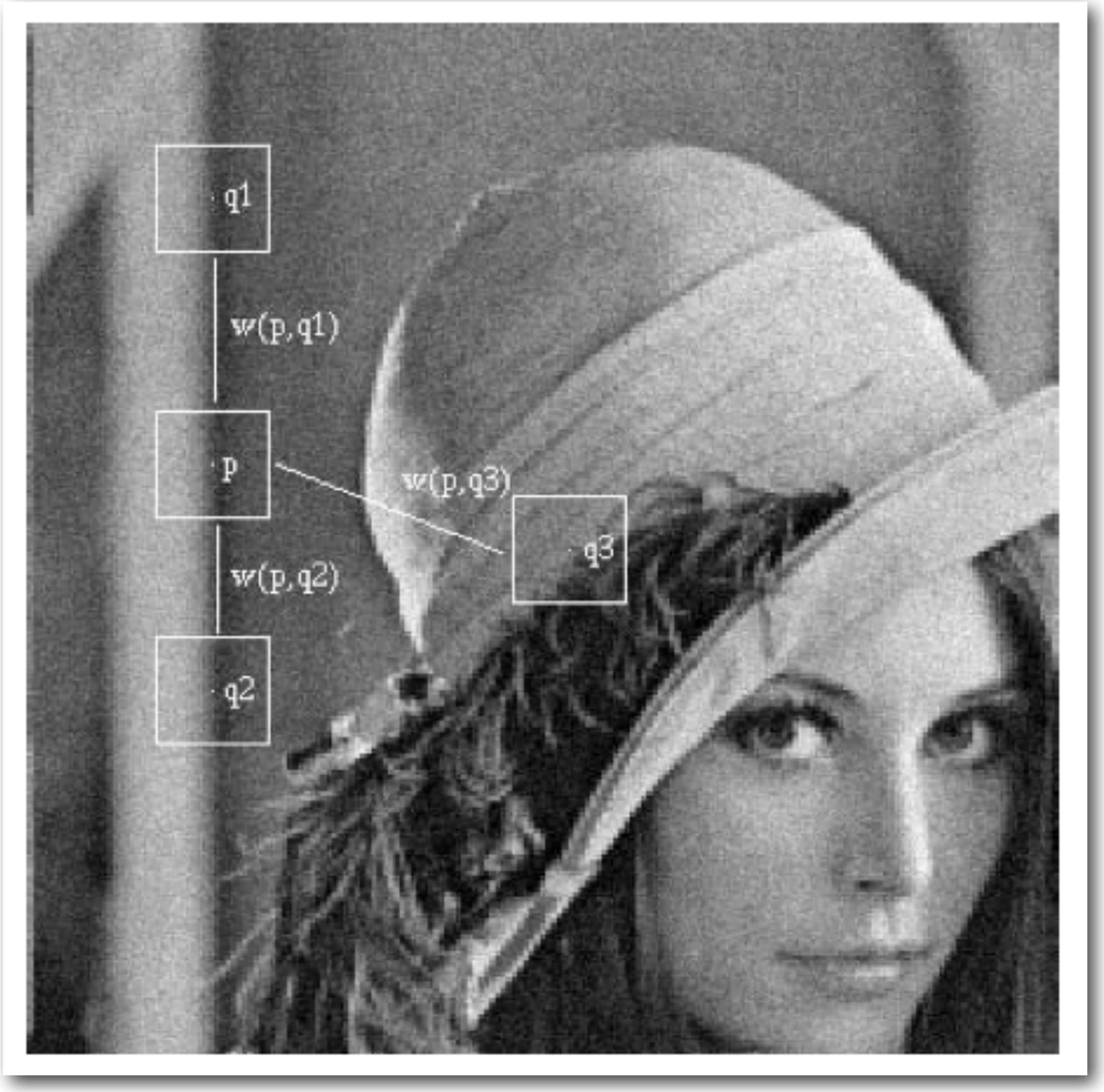
Denoising using non-local means

- **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)

Color-space conversion

- **Measurements of sensor depend on sensor's spectral response**
  - Response depends on bandwidths filtered by color filter array
- **Convert representation to sensor-independent basis: e.g., sRGB**
  - 3 x 3 matrix multiplication

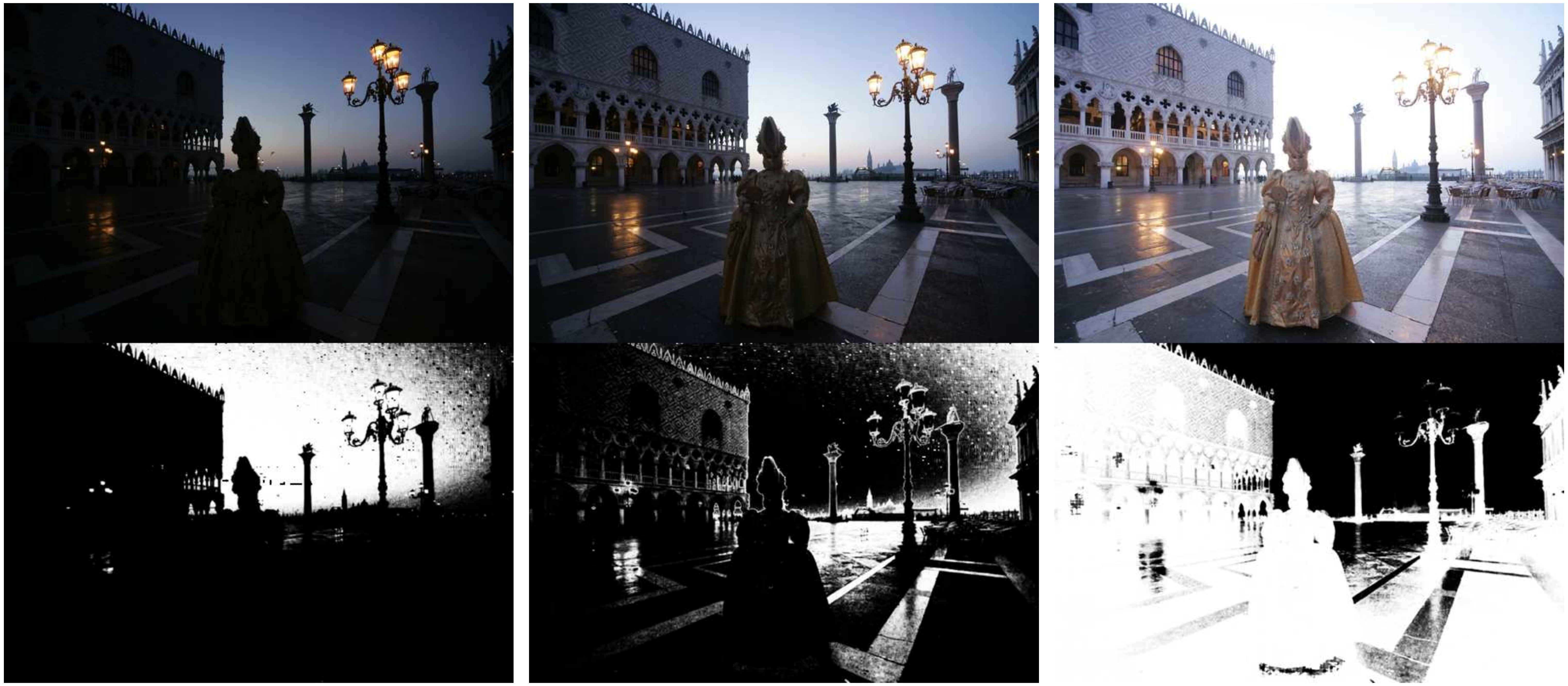
```
output_rgb_pixel = COLOR_CONVERSION_MATRIX * input_rgb_pixel
```

Represented in standard color space (e.g., RGB)

Represented in sensor-specific basis

Note: modern pipelines may perform more sophisticated (and non-linear) color transformations at this point. e.g., use a big lookup table to adjust certain tones (e.g., make sky-like tones bluer)

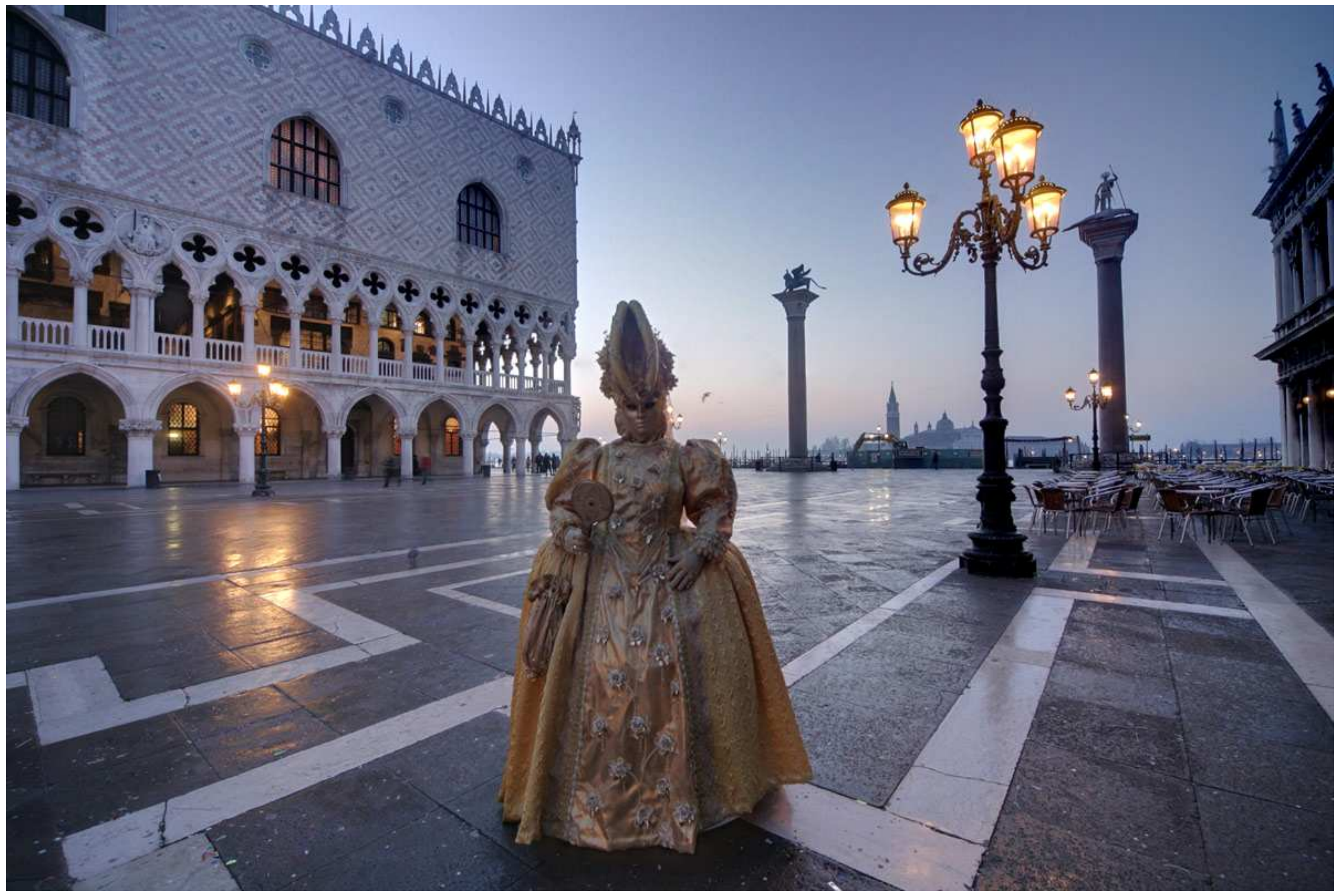
Local-tone adjustment



Weights

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

(more details in later lecture)



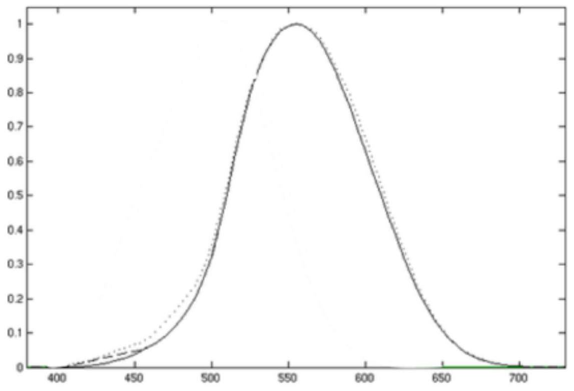
Combined image
(unique weights per pixel)

Image credit: Mertens 2007

Lightness (perceived brightness) aka luma

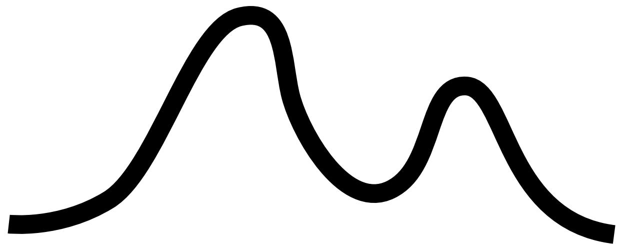
Lightness (L*) $\xleftarrow{?}$ Luminance (Y) $= \int_\lambda$

(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: $L^* \propto Y^{0.5}$

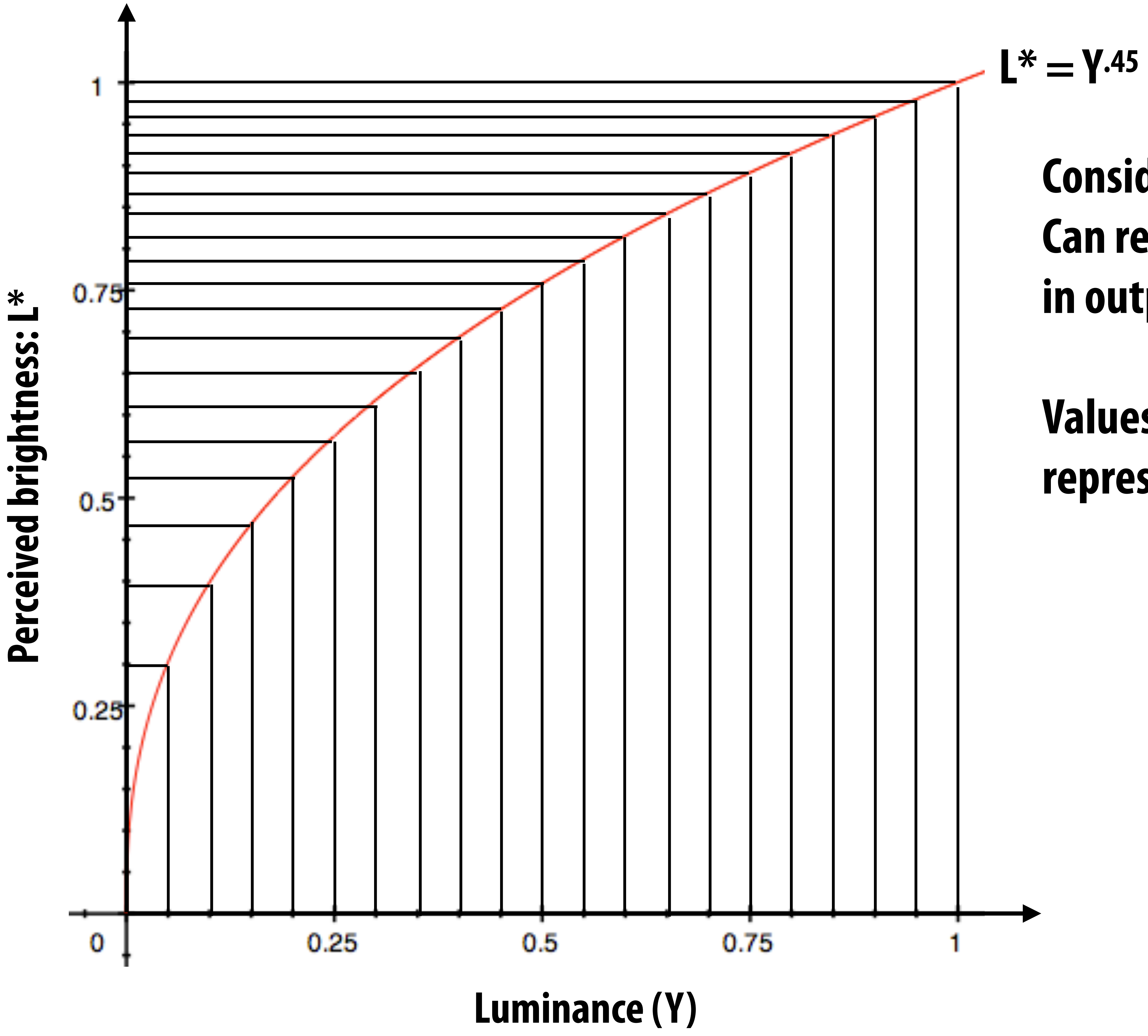
In a dark room, you turn on a light with luminance: Y_1

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)



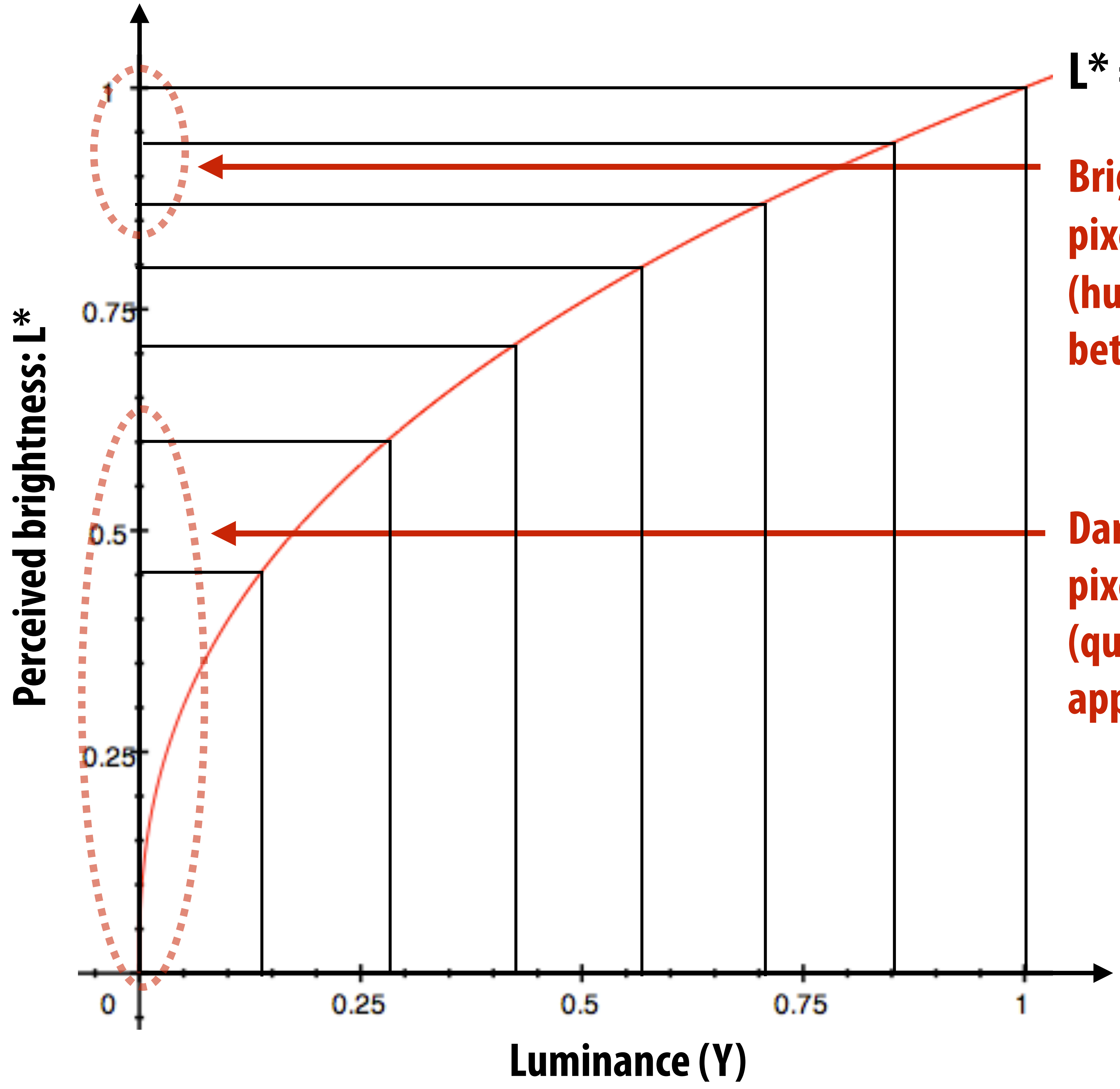
Consider 12-bit sensor pixel:
Can represent 4096 unique luminance values in output image

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



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

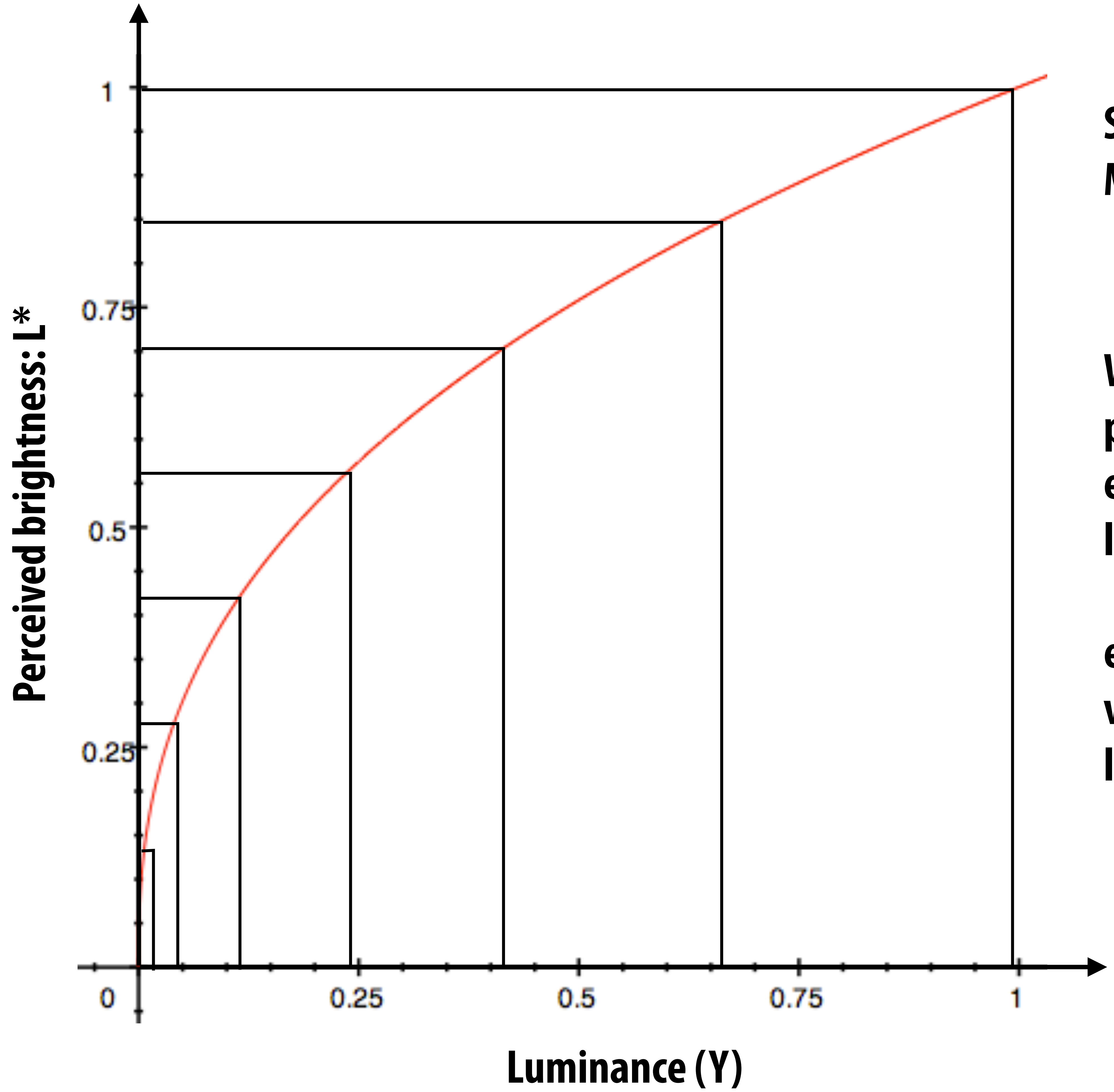
Bright regions of image: perceived difference between pixels that differ by one step in luminance is small! (human may not even be able to perceive difference between pixels that differ by one step in luminance!)

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, 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?

Summary: simplified image processing pipeline

- Correct for sensor bias (using measurements of optically black pixels)
- Correct pixel defects
- Vignetting compensation
- Dark-frame subtract (optional)
- White balance

12-bits per pixel
1 intensity per pixel
Pixel values linear in energy

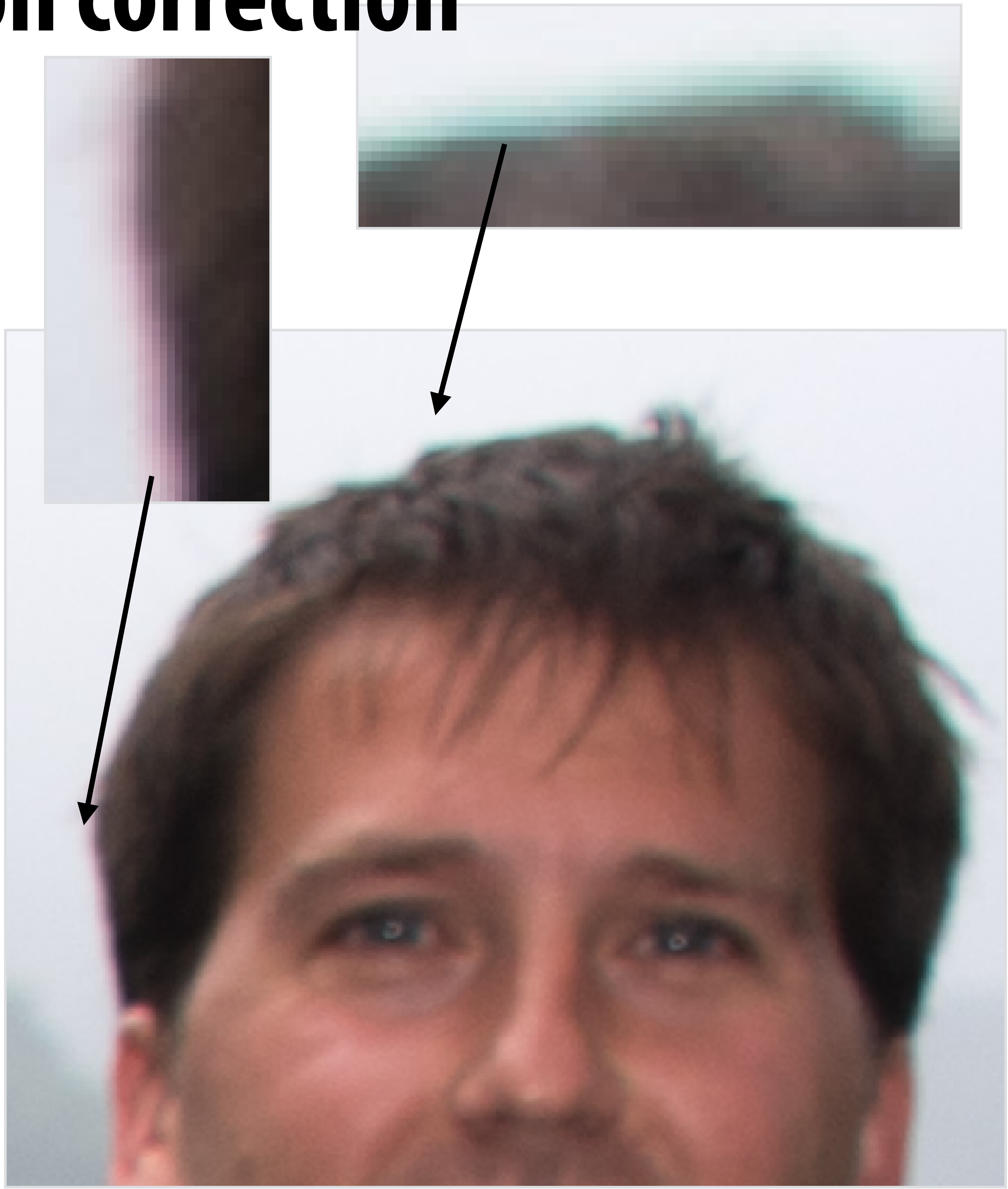
- Demosaic
- Denoise / sharpen, etc.
- Color Space Conversion

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

- Local tone mapping
- Gamma Correction (non-linear mapping)
- Color Space Conversion (Y'CbCr)

3x8-bits per pixel
Pixel values perceptually linear

Chromatic aberration correction



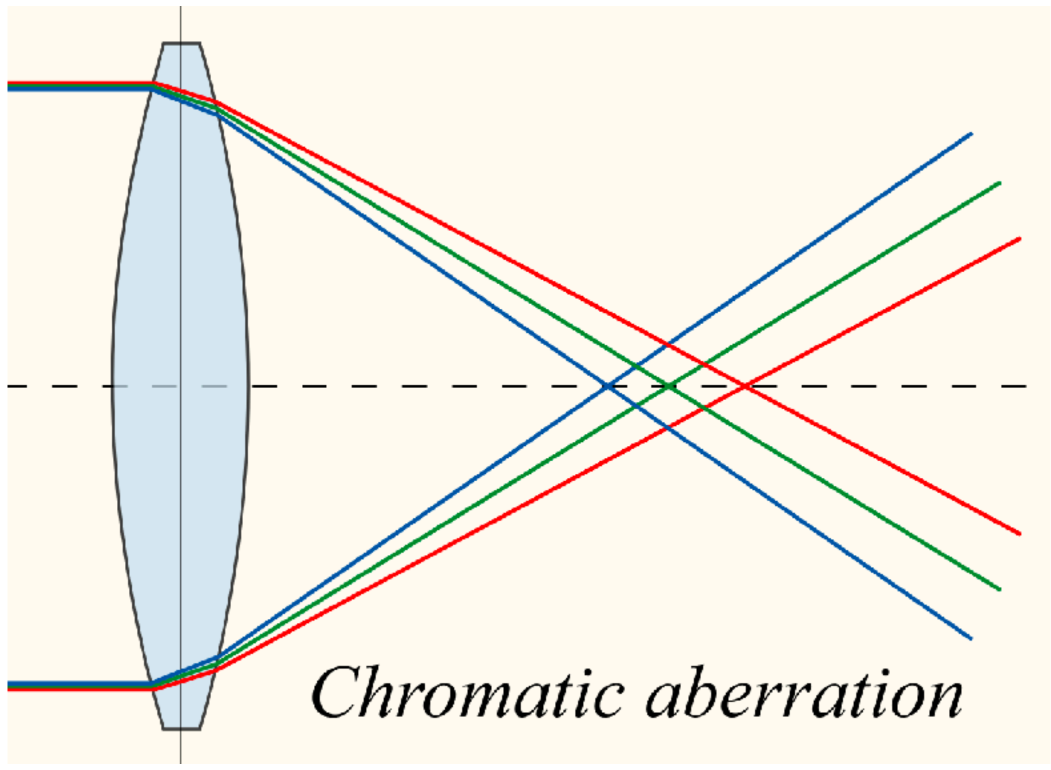


Image credit: Wikipedia

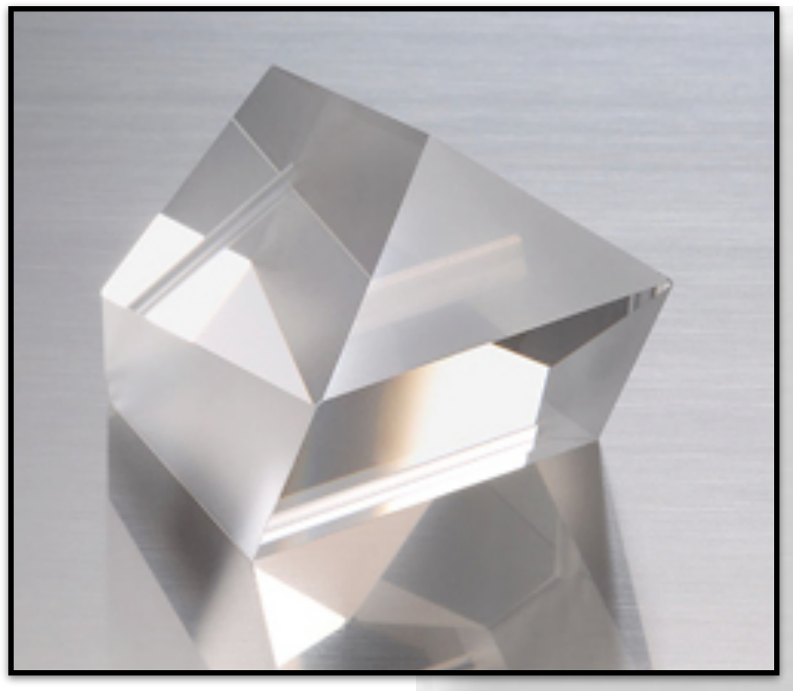
Autofocus demos

- Phase-detection auto focus
  - Common in SLRs

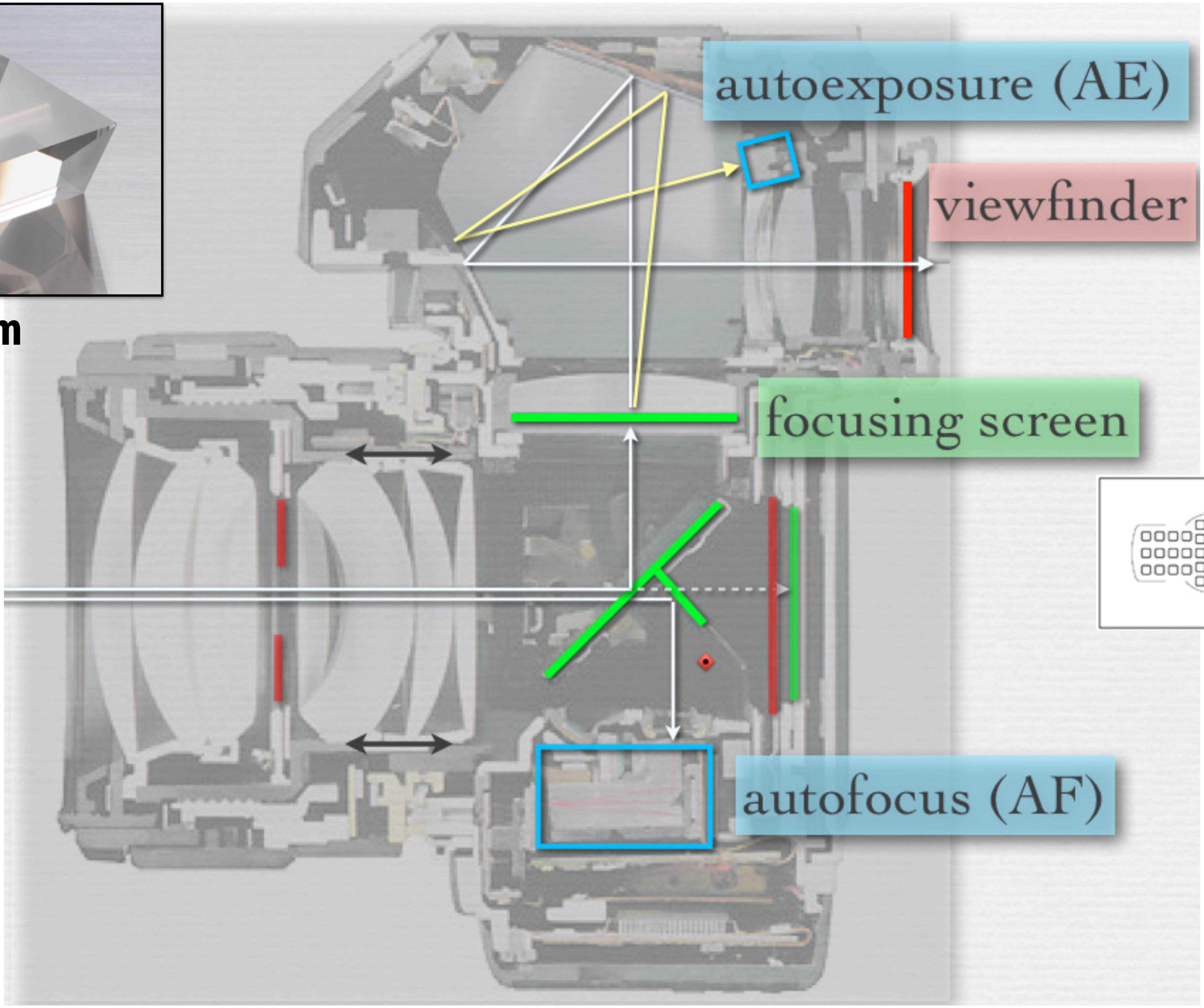
- Contrast-detection auto focus
  - Point-and-shoots, smart-phone cameras

Demo credits: Marc Levoy and Stanford CS178 course staff

SLR Camera



Pentaprism



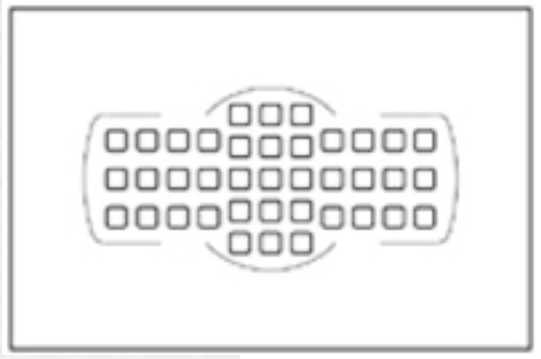
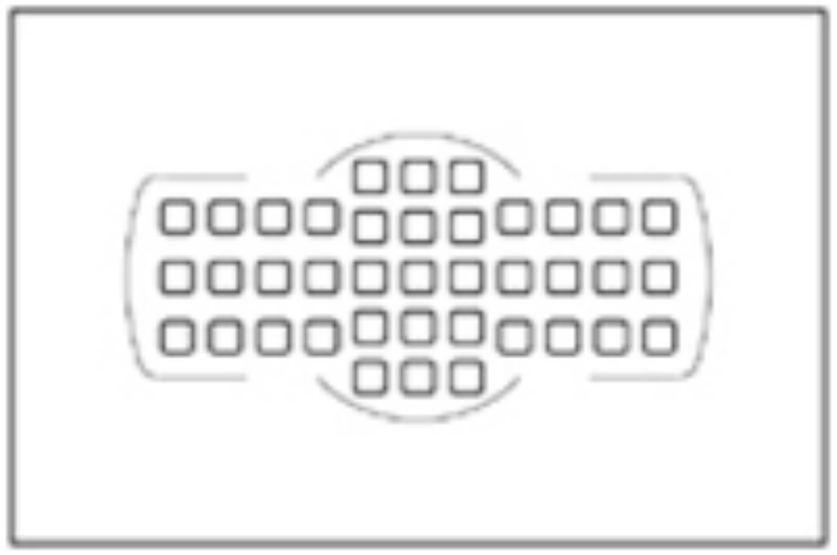
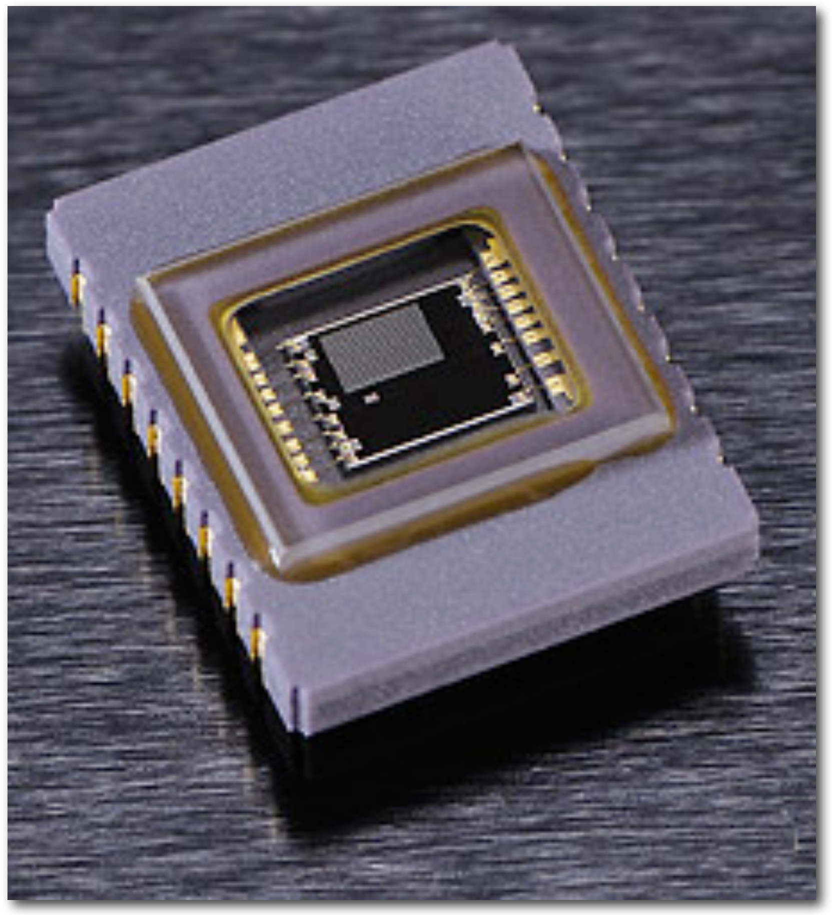


Image credits: Nikon, Marc Levoy

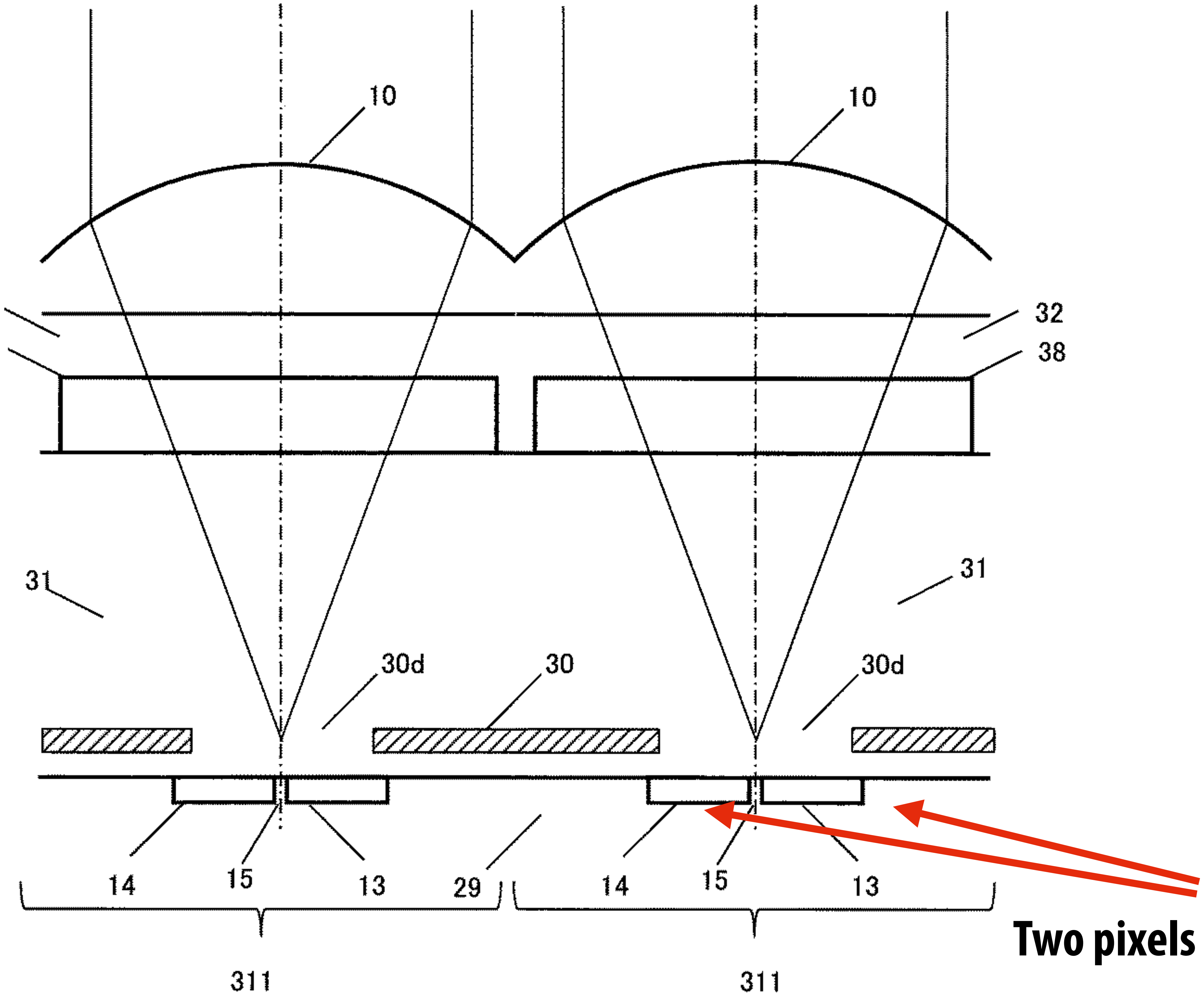
Nikon D7000

- Auto-focus sensor: 39 regions
- Metering sensor: 2K pixels
  - Auto-exposure
  - Auto-white-balance
  - Subject tracking to aid focus (predicts movement)
- Shutter lag ~ 50ms





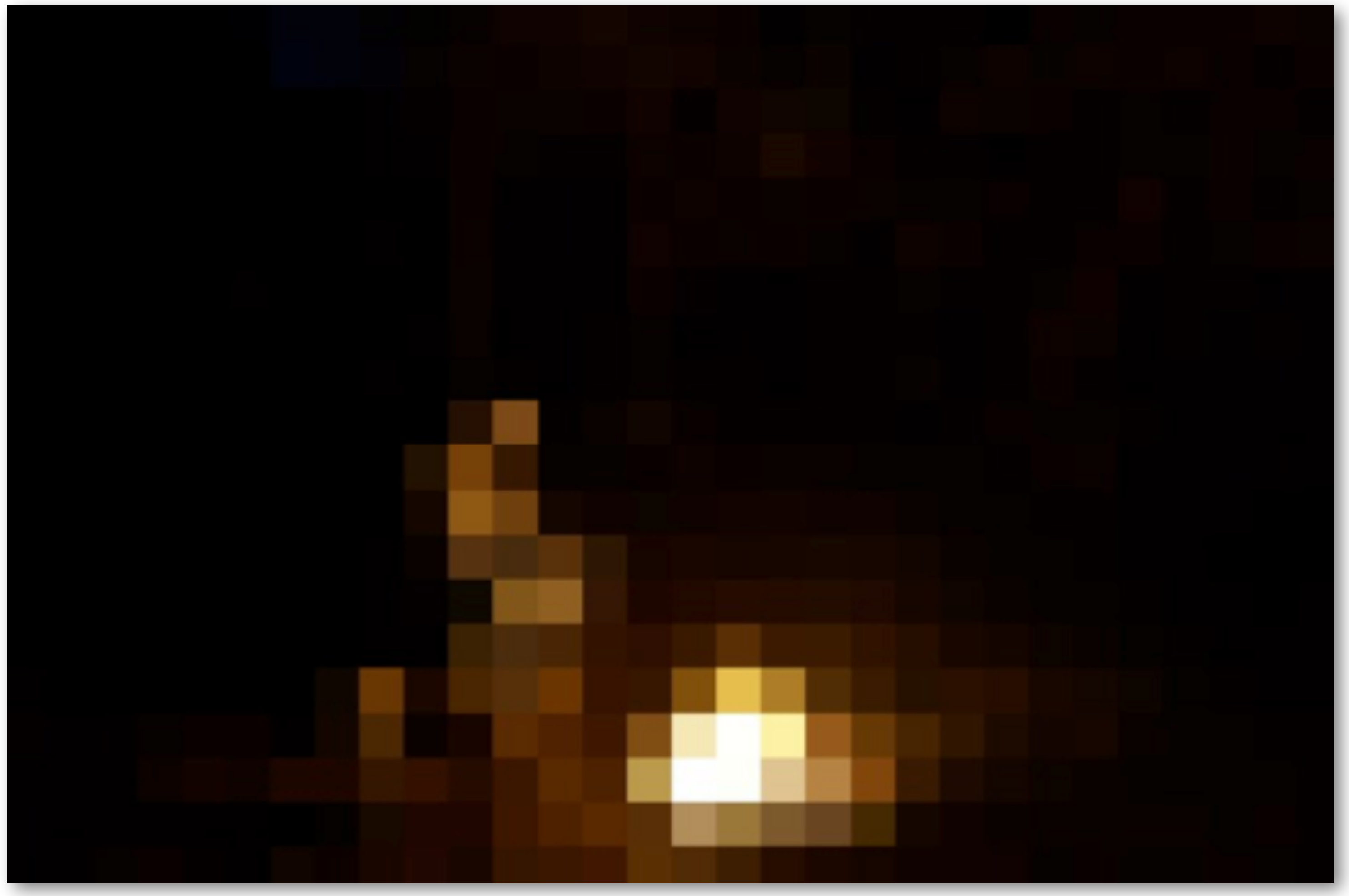
Split pixel sensor



Two pixels under microlens

Image credit: Nikon

Auto exposure



Low resolution metering sensor capture

Metering sensor pixels are large
(higher dynamic range than main sensor)

How do we set exposure?

What if a camera doesn't have a separate metering sensor?

Image credits: Marc Levoy, Andrew Adams

AF/AE summary

- DSLRs have additional sensing/processing hardware to assist with the "3A's" (auto-focus, auto-exposure, auto-white-balance)
  - Phase-detection AF: optical system directs light to AF sensor
  - Example: Nikon metering sensor: large pixels to avoid over-saturation

- Point-and-shoots/smartphone cameras make these measurements by performing image processing operations on data from the main sensor
  - Contrast-detection AF: search for lens position that produces large image gradients
  - Exposure metering: if pixels are saturating, meter again with lower exposure

- In general, implementing AF/AE/AWB is an image understanding problem ("computer vision")
  - Understand the scene well enough to set the camera's image capture and image processing parameters to best approximate the image a human would perceive
  - As processing/sensing capability increases, algorithms are becoming more sophisticated

Takeways

- The values of pixels in picture you see on screen are quite different than the values output by the photosensor in a modern digital camera.

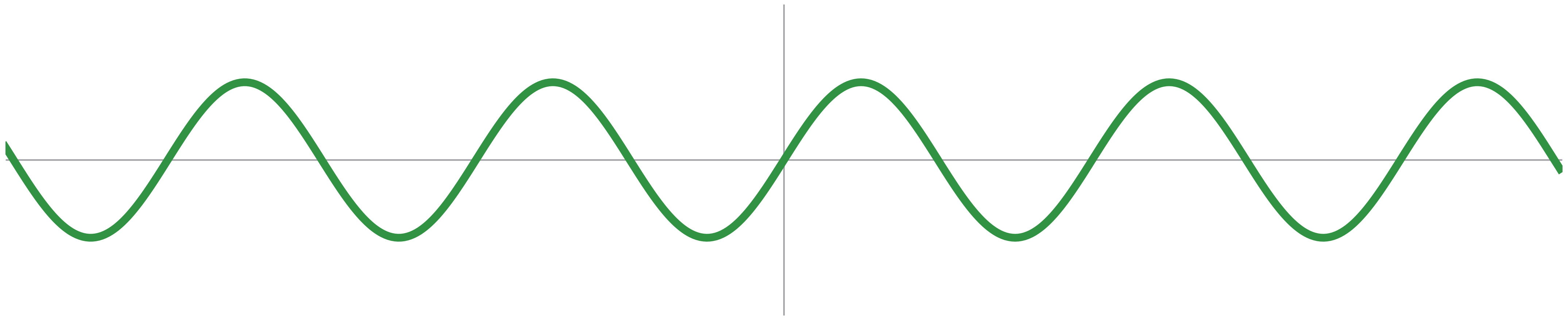
- The sequence of operations we discussed today is carried out at high frame rates by special purpose hardware in most cameras today

- In the coming lectures we'll discuss more advanced image processing operations in use in modern camera pipelines
  - Local contrast enhancement, advanced denoising, high-dynamic range imaging, etc.
  - Growing sophistication and diversity of techniques suggests that image-signal processors will likely become increasingly programmable (e.g., Qualcomm's Hexagon DSP)

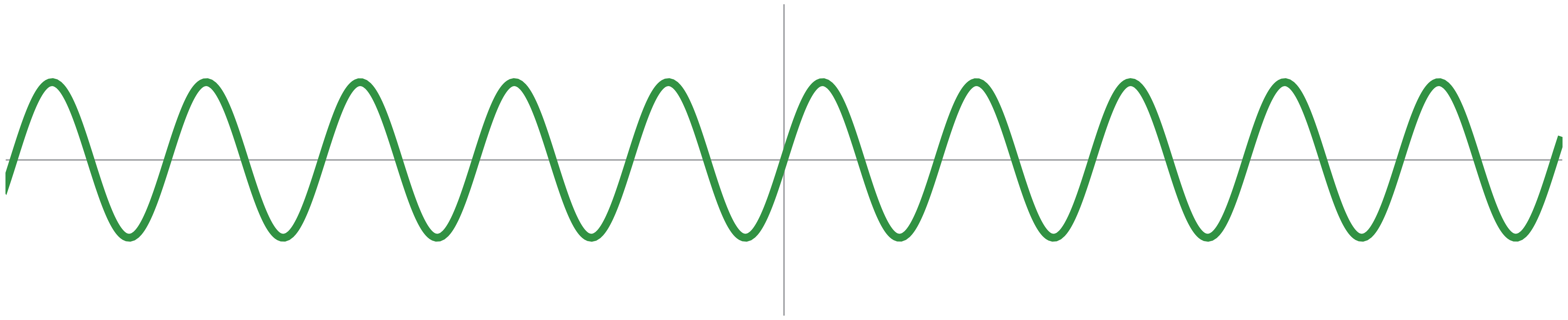
Frequency interpretation of images

Representing sound as a superposition of frequencies

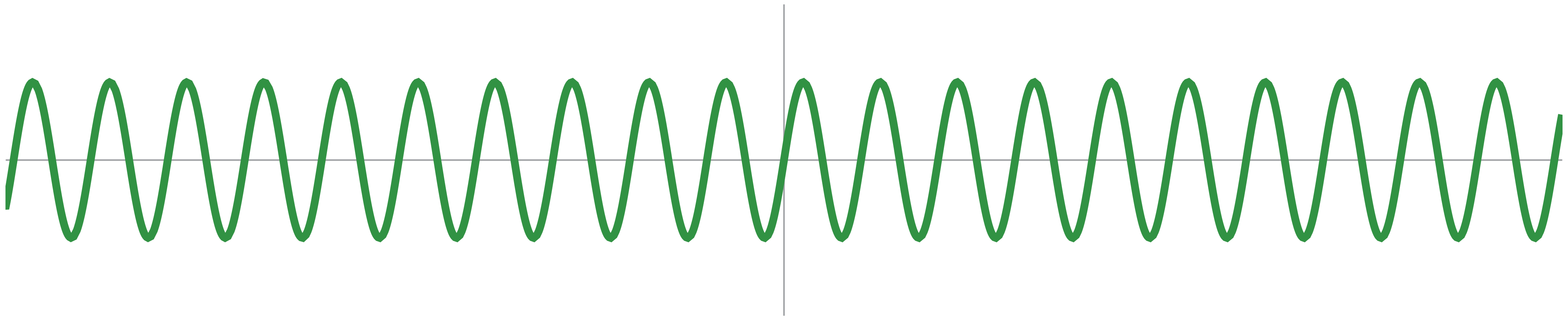
$f_1(x) = sin(\pi x)$



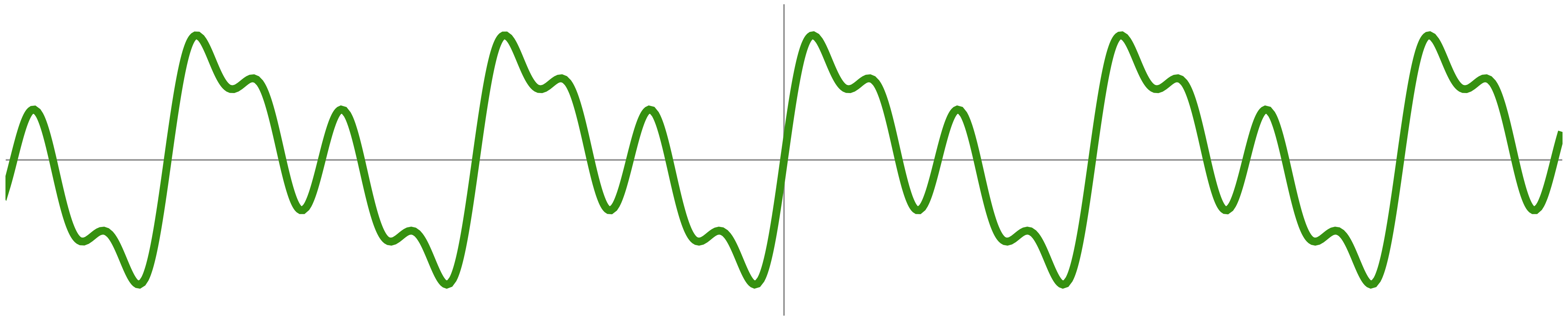
$f_2(x) = sin(2\pi x)$



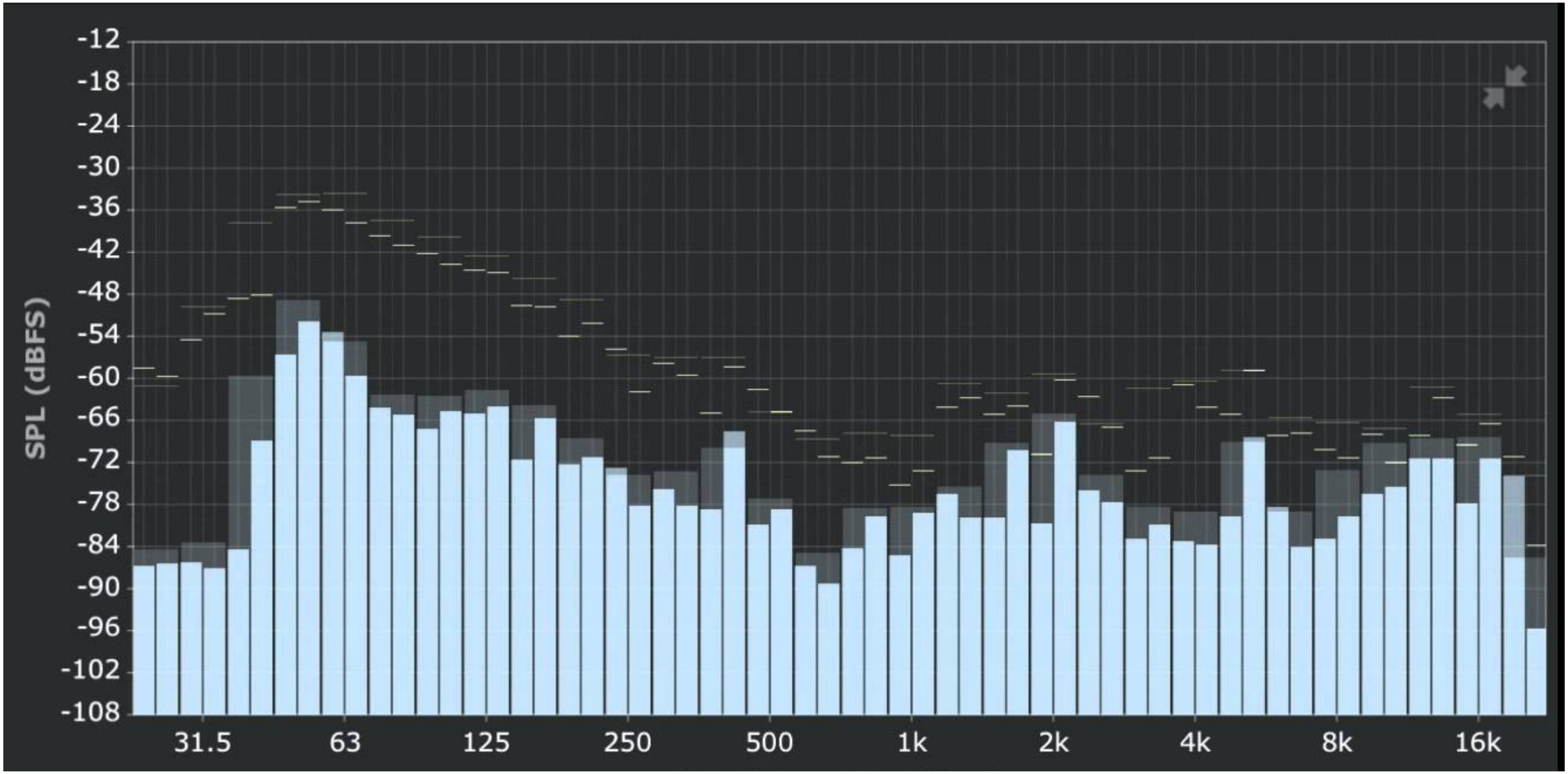
$f_4(x) = sin(4\pi x)$



$f(x) = f_1(x) + 0.75\, f_2(x) + 0.5\, f_4(x)$



Audio spectrum analyzer: representing sound as a sum of its constituent frequencies



Intensity of low-frequencies (bass)

Intensity of high frequencies

Image credit: ONYX Apps

Fourier transform

- **Convert representation of signal from spatial/temporal domain to frequency domain by projecting signal into its component frequencies**

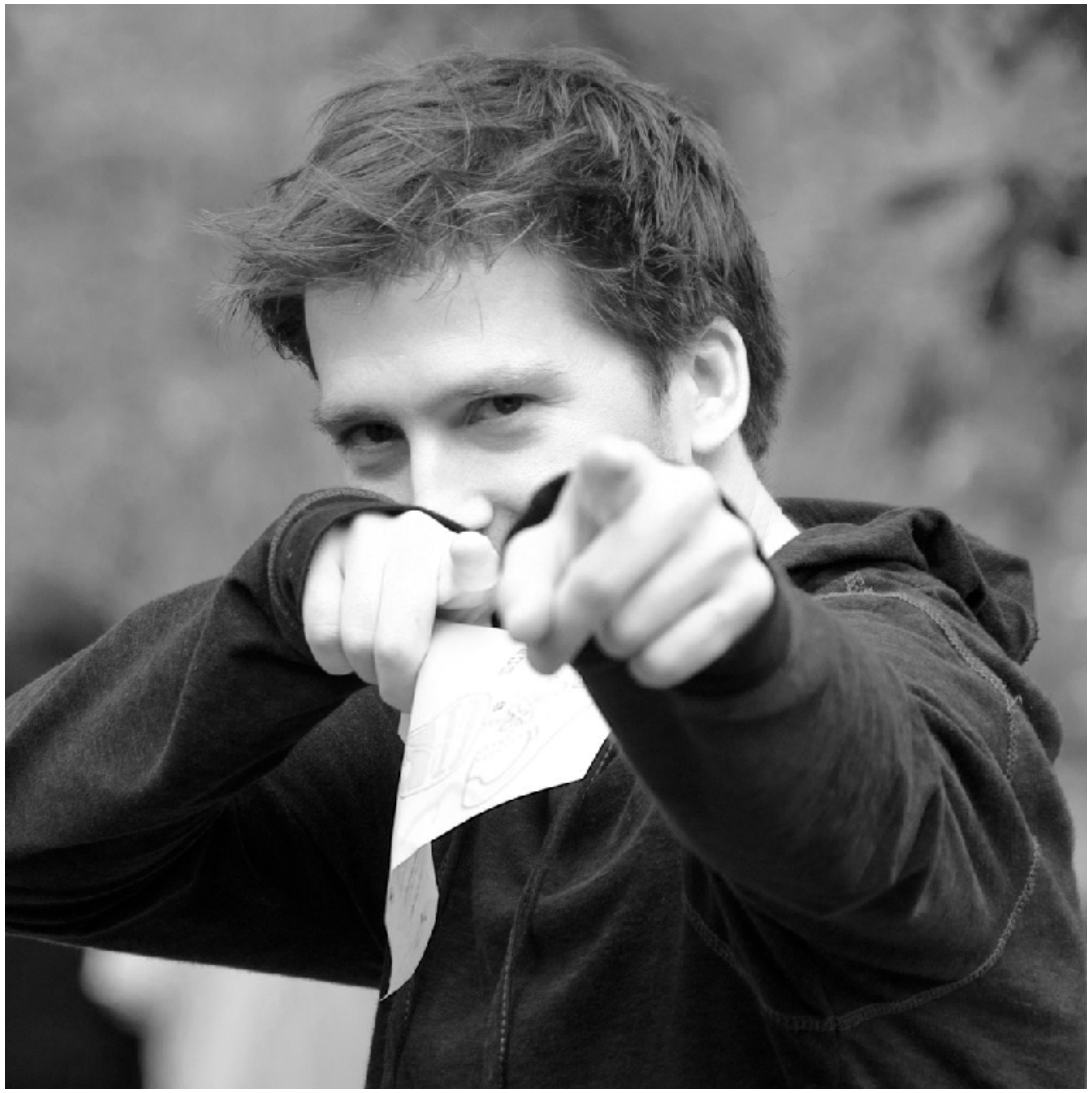
$$f(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \xi} dx$$

$$= \int_{-\infty}^{\infty} f(x)(cos(2\pi \xi x) - sin(2\pi \xi x))dx$$

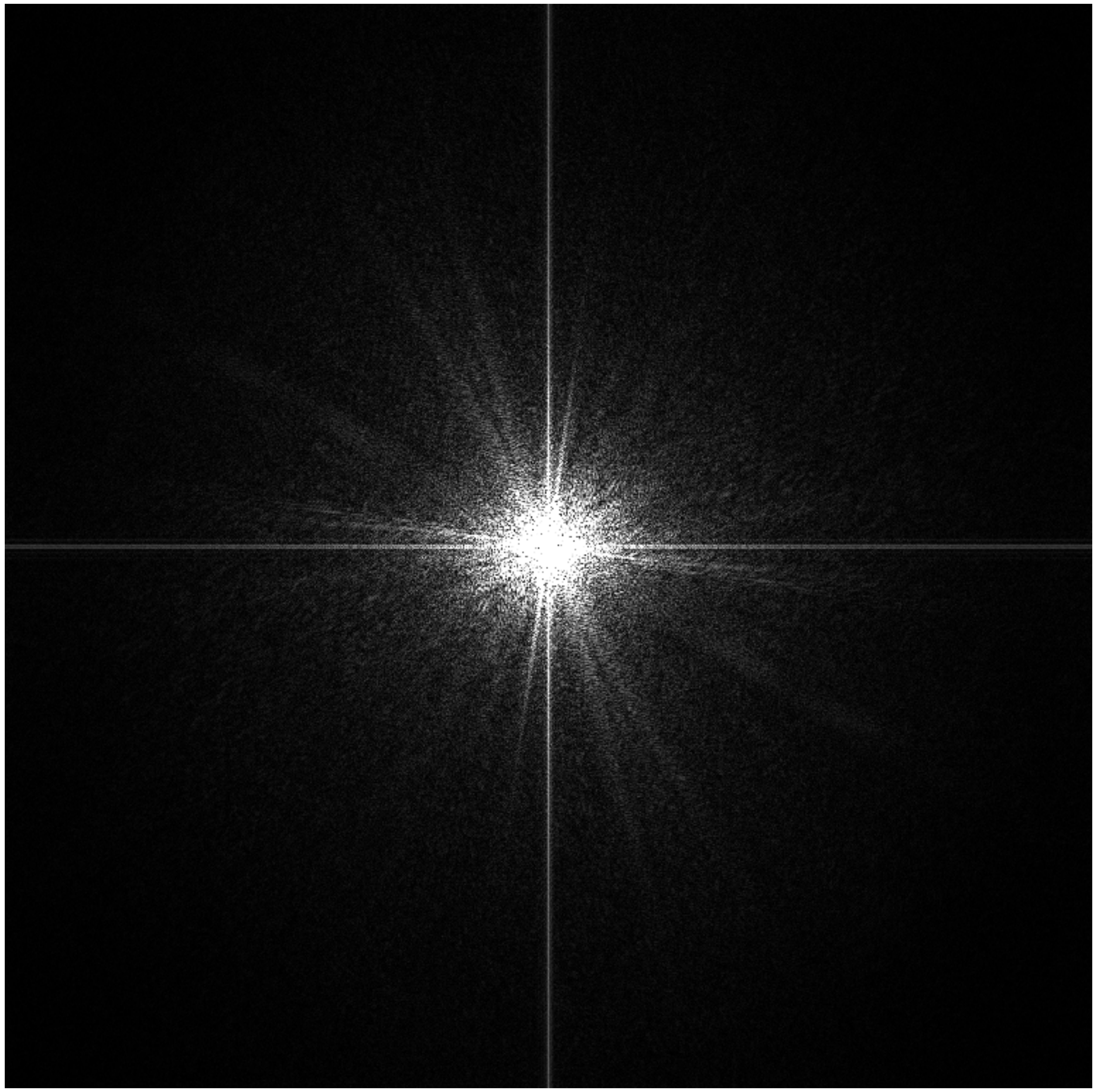
- **2D form:**

$$f(u,v) = \int\int f(x,y) e^{-2\pi i(ux+vy)} dxdy$$

Visualizing the frequency content of images

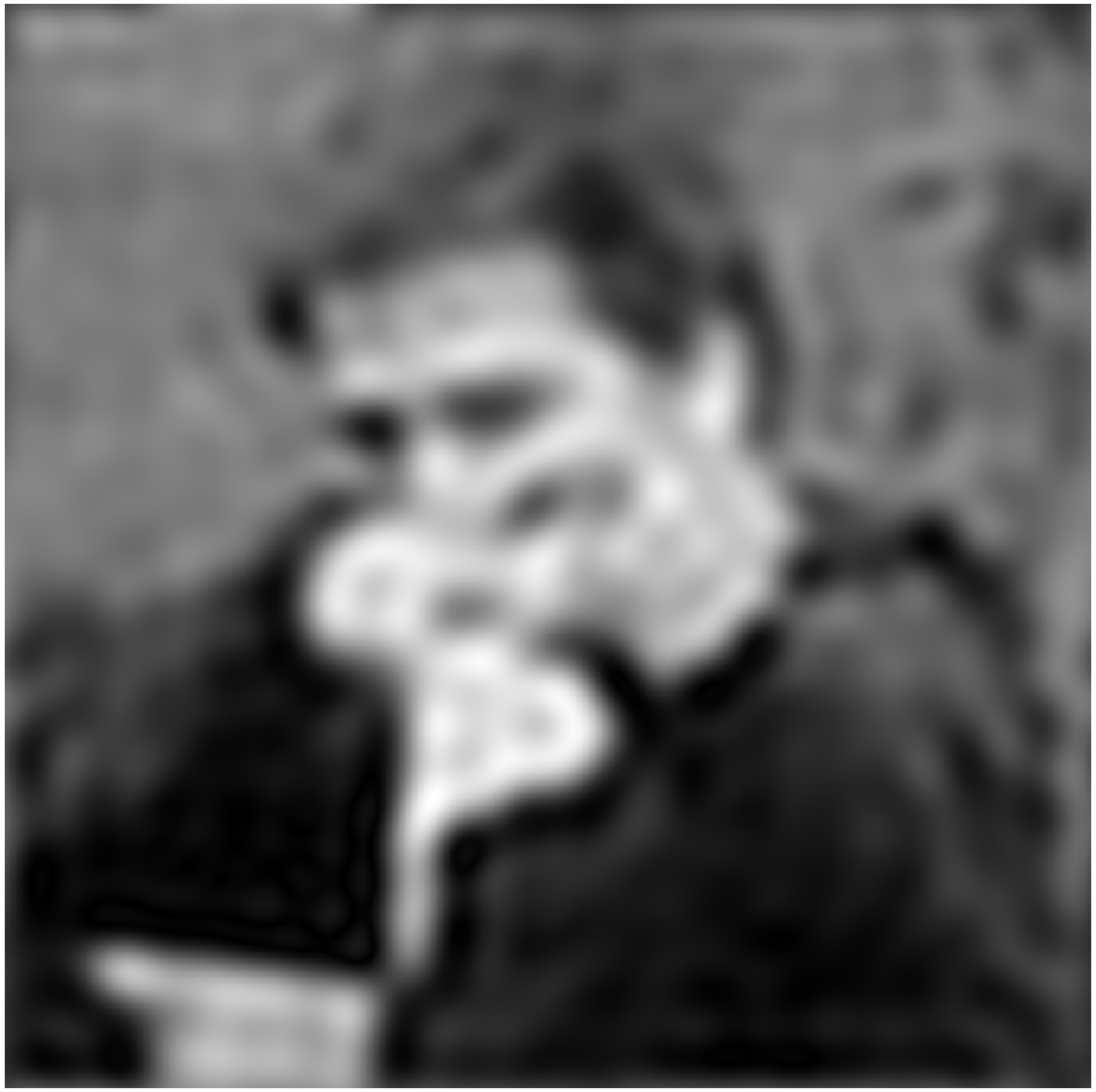


Spatial domain result

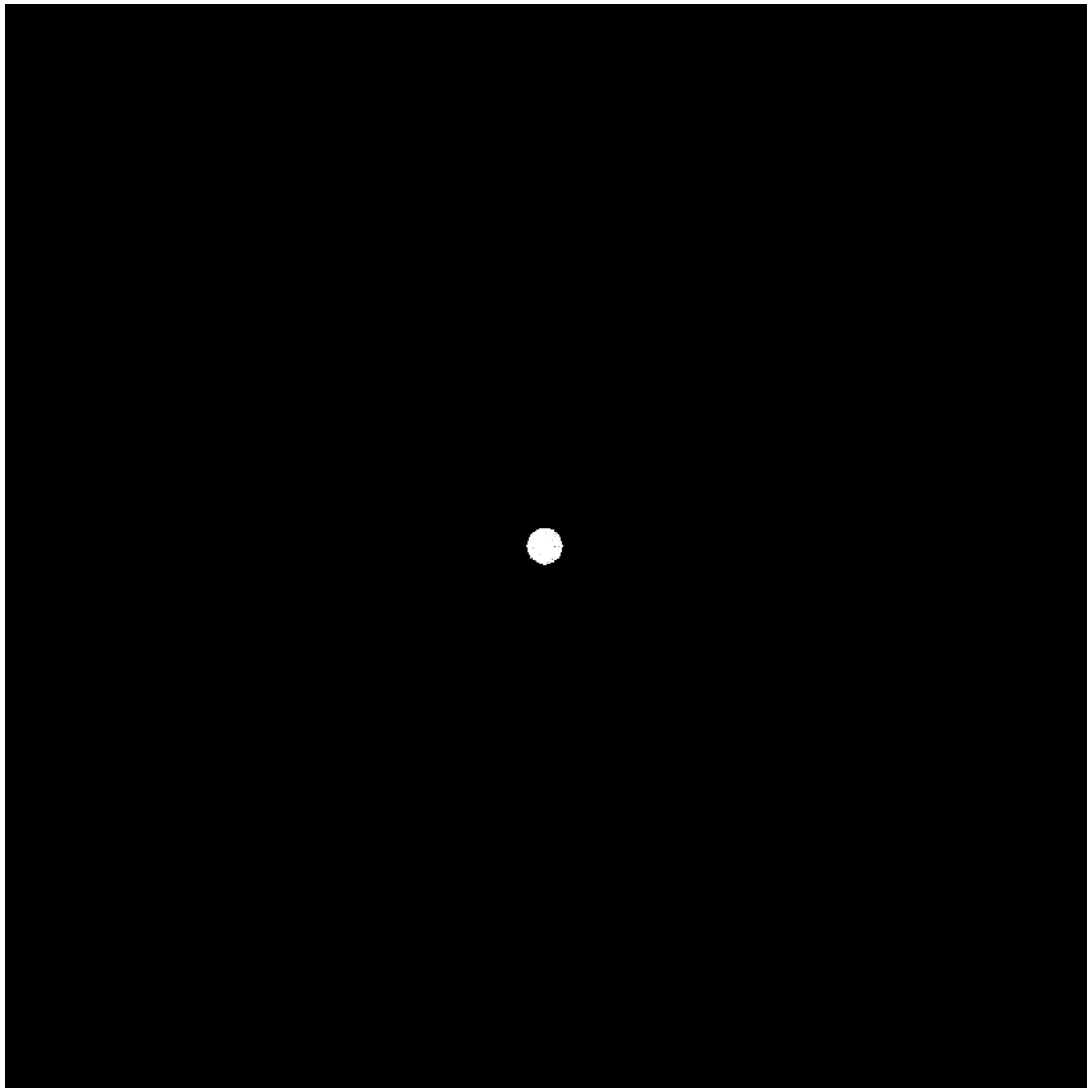


Spectrum

Low frequencies only (smooth gradients)

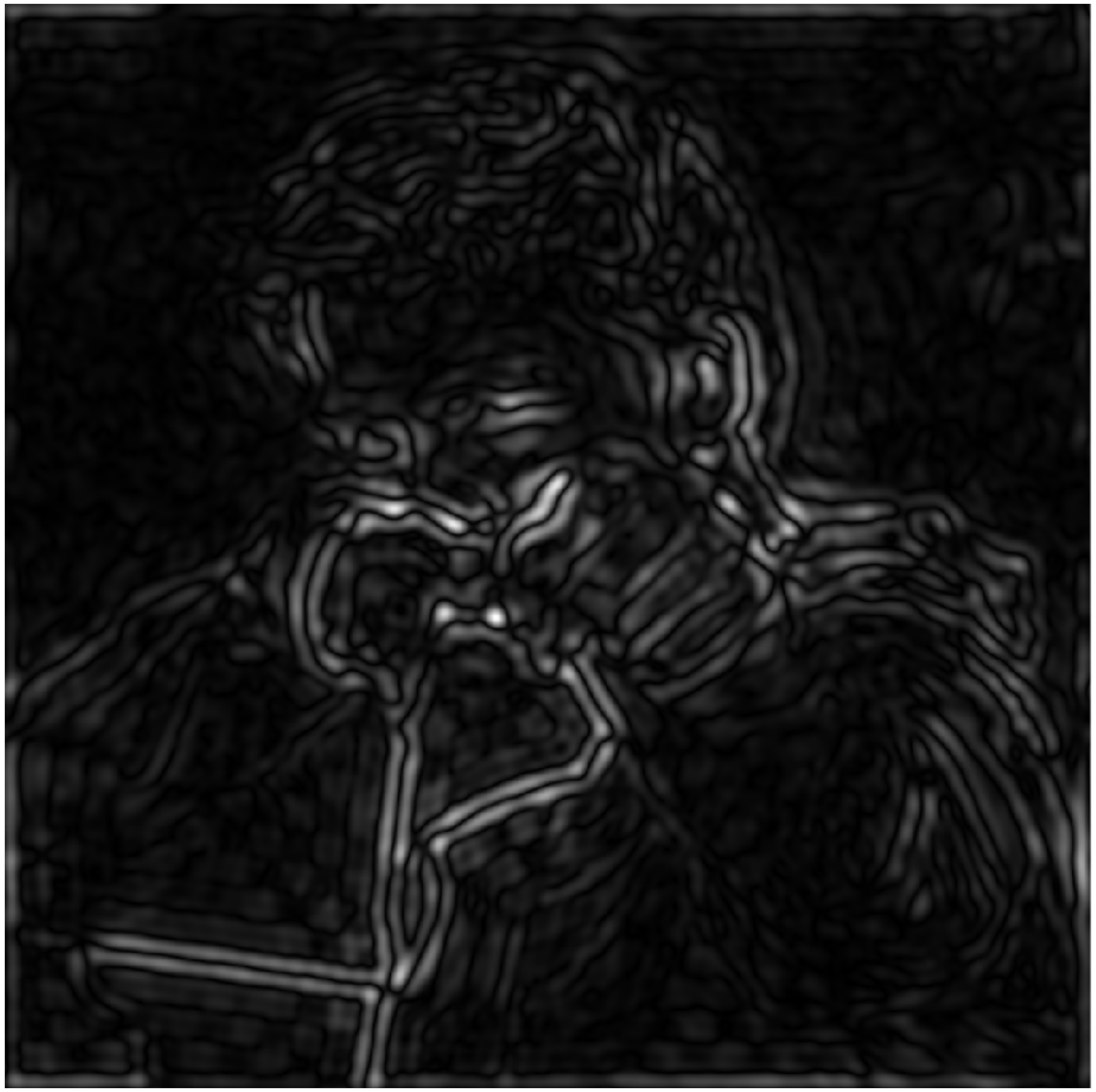


Spatial domain result

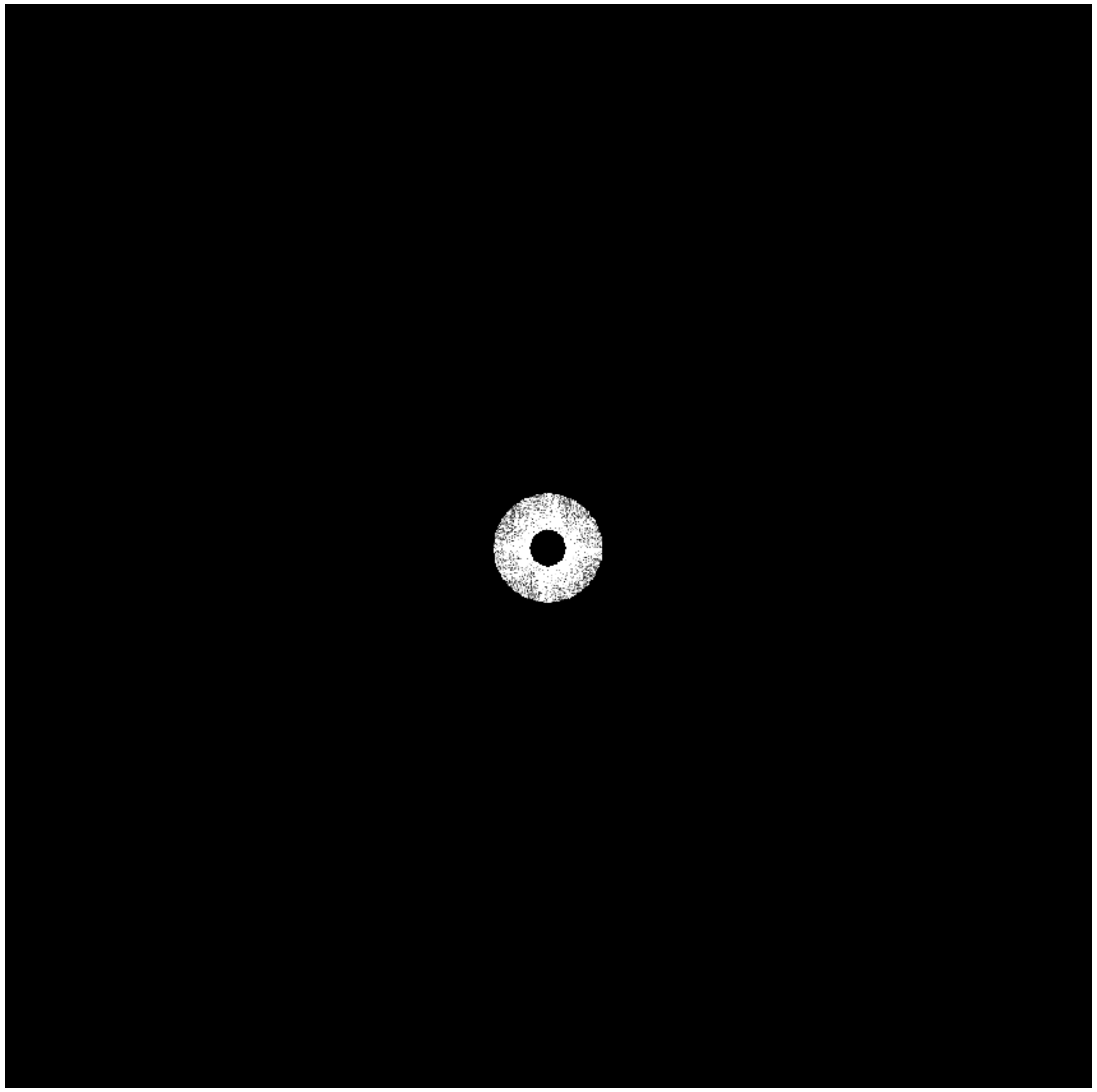


Spectrum (after low-pass filter)
All frequencies above cutoff have 0 magnitude

Mid-range frequencies

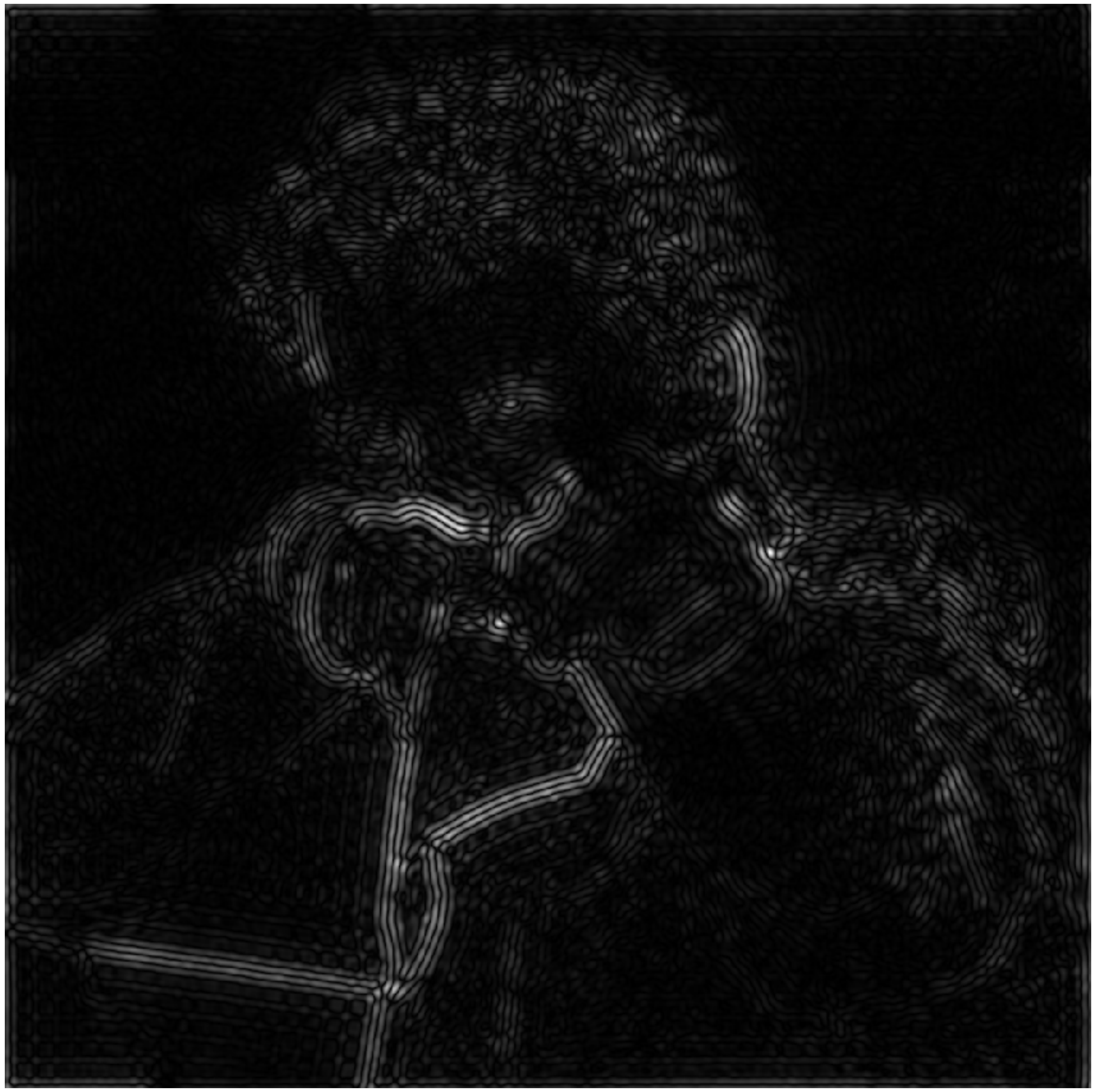


Spatial domain result

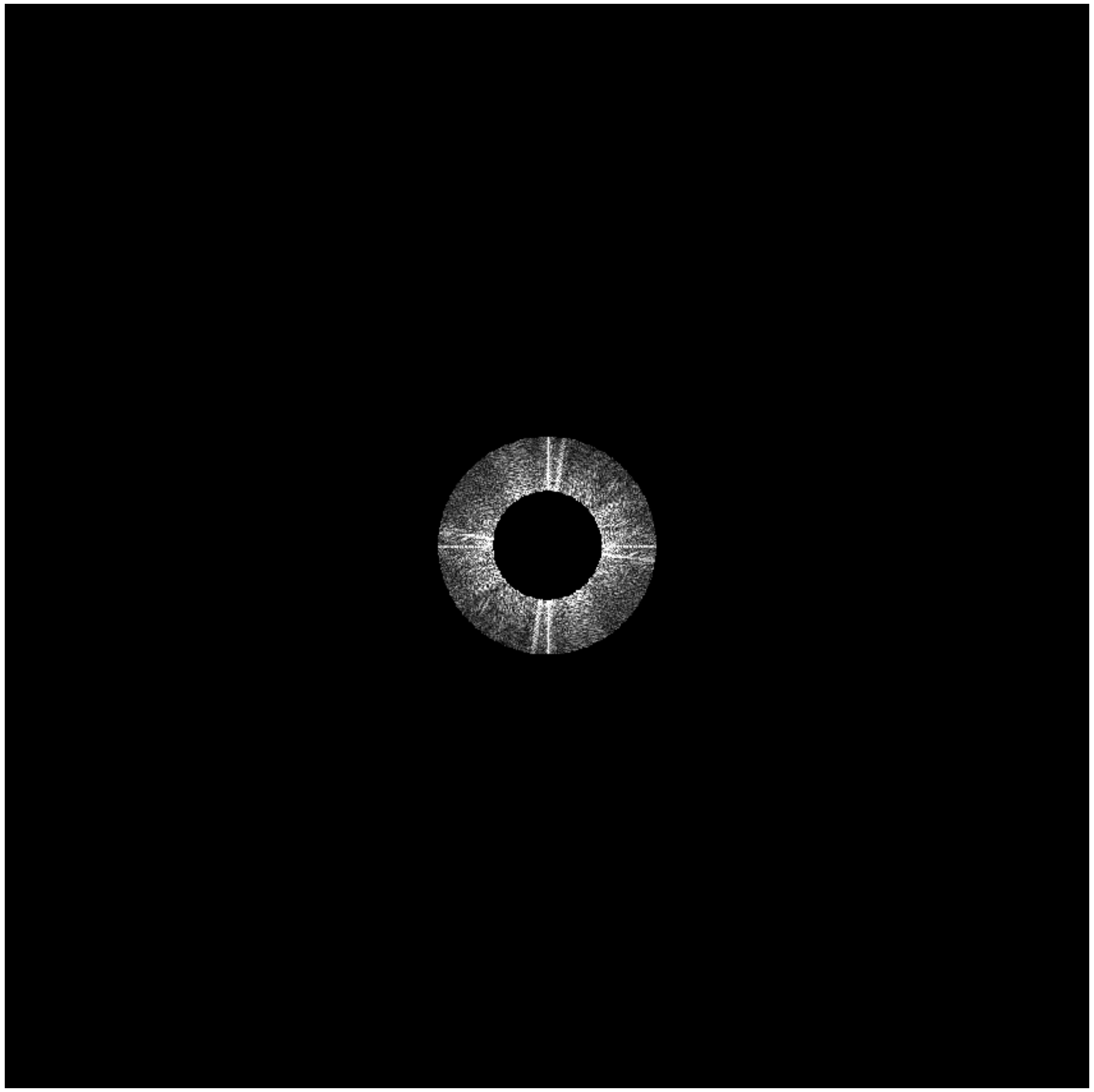


Spectrum (after band-pass filter)

Mid-range frequencies

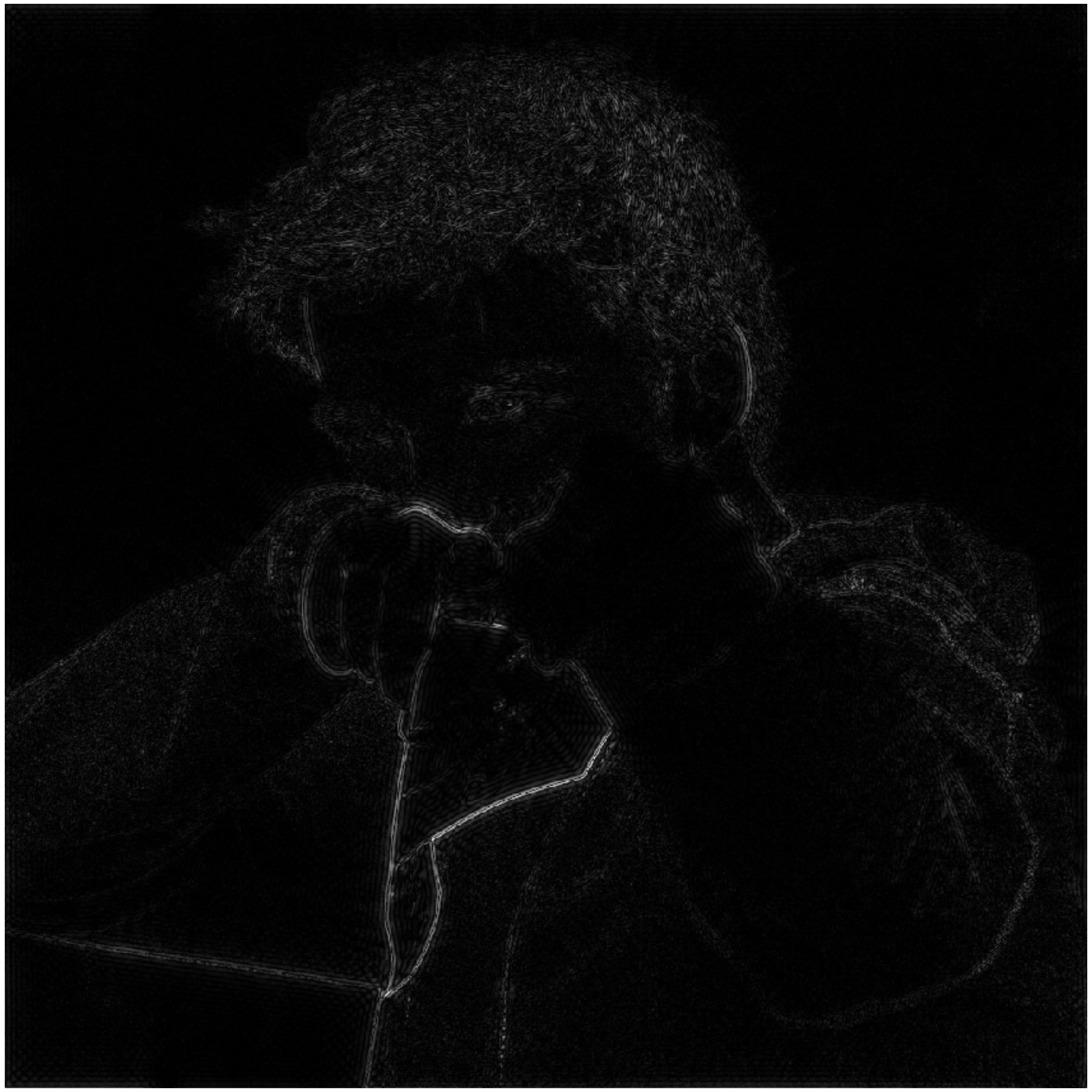


Spatial domain result

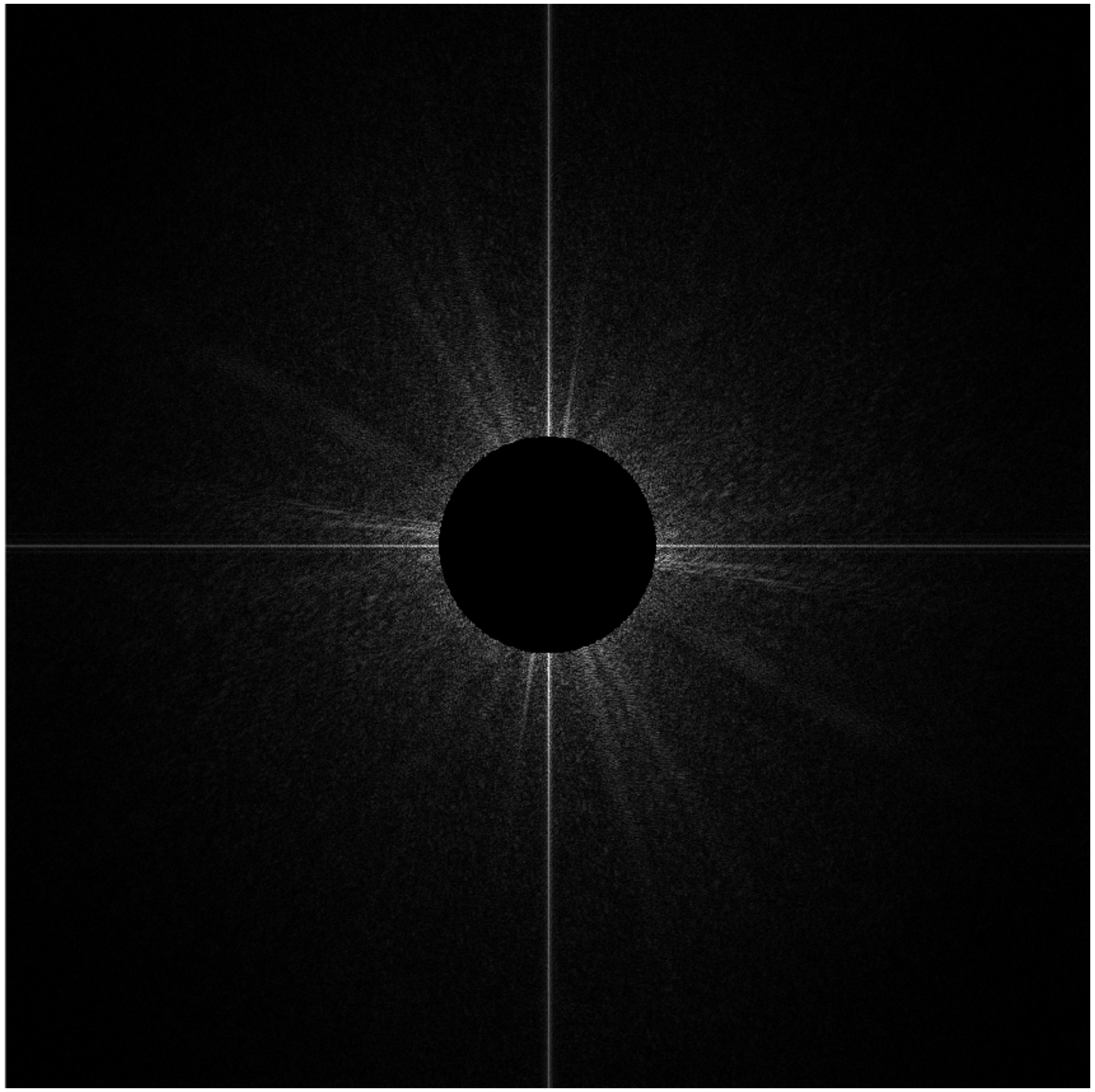


Spectrum (after band-pass filter)

High frequencies (edges)

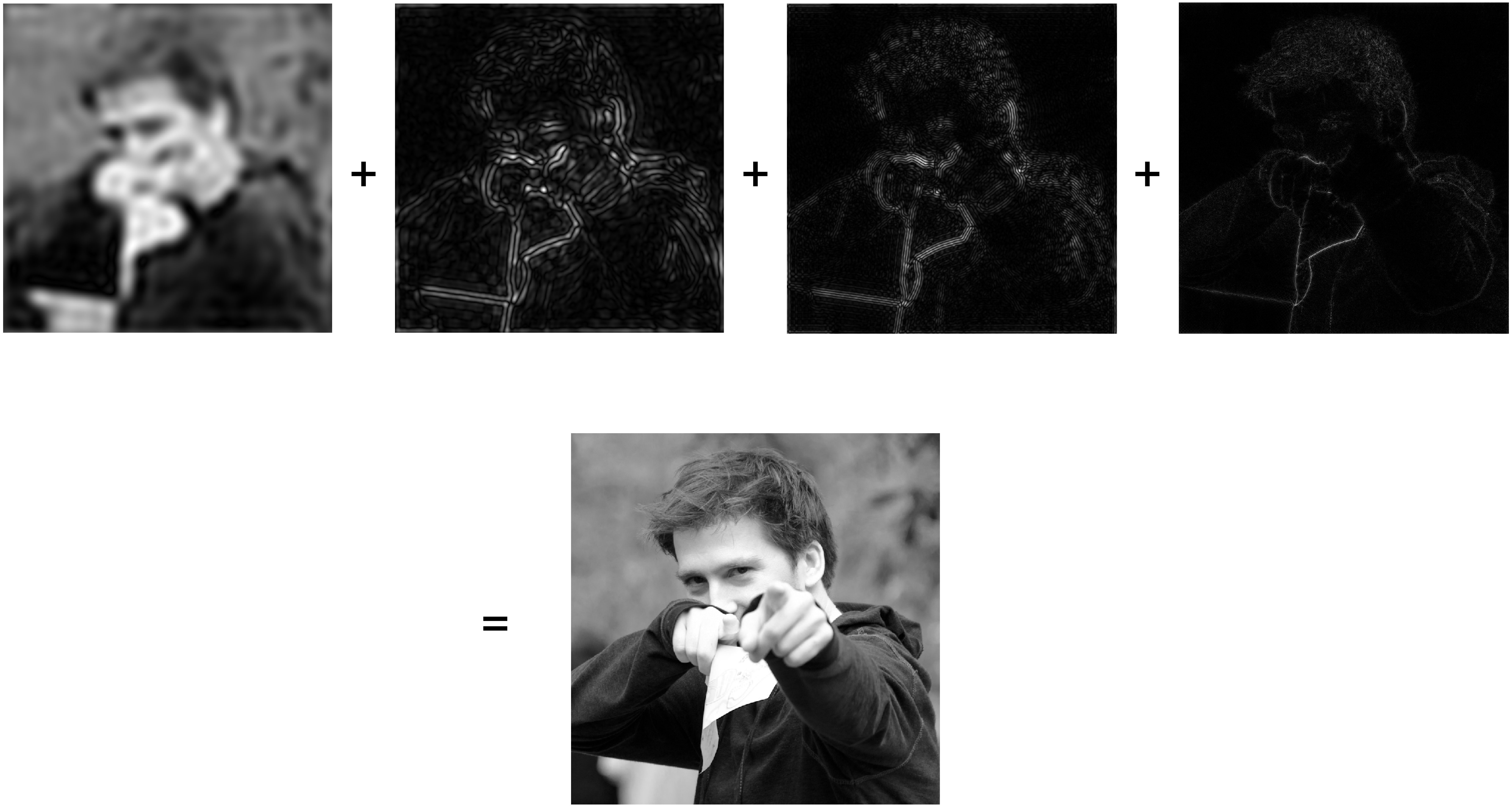


Spatial domain result
(strongest edges)



Spectrum (after high-pass filter)
All frequencies below threshold
have 0 magnitude

An image as a sum of its frequency components



Another (linear) sharpening filter

$$\text{blurred} = g * I$$

$$\text{fine} = I - \text{blurred}$$

Extract high frequencies

$$\text{sharpened} = I + 0.5 \times \text{fine}$$

Boost high frequencies

But what if we'd like to localize image edits both in space and in frequency?

Downsample

- Step 1: Remove high frequencies
- Step 2: Sparsely sample pixels (below: every other pixel)

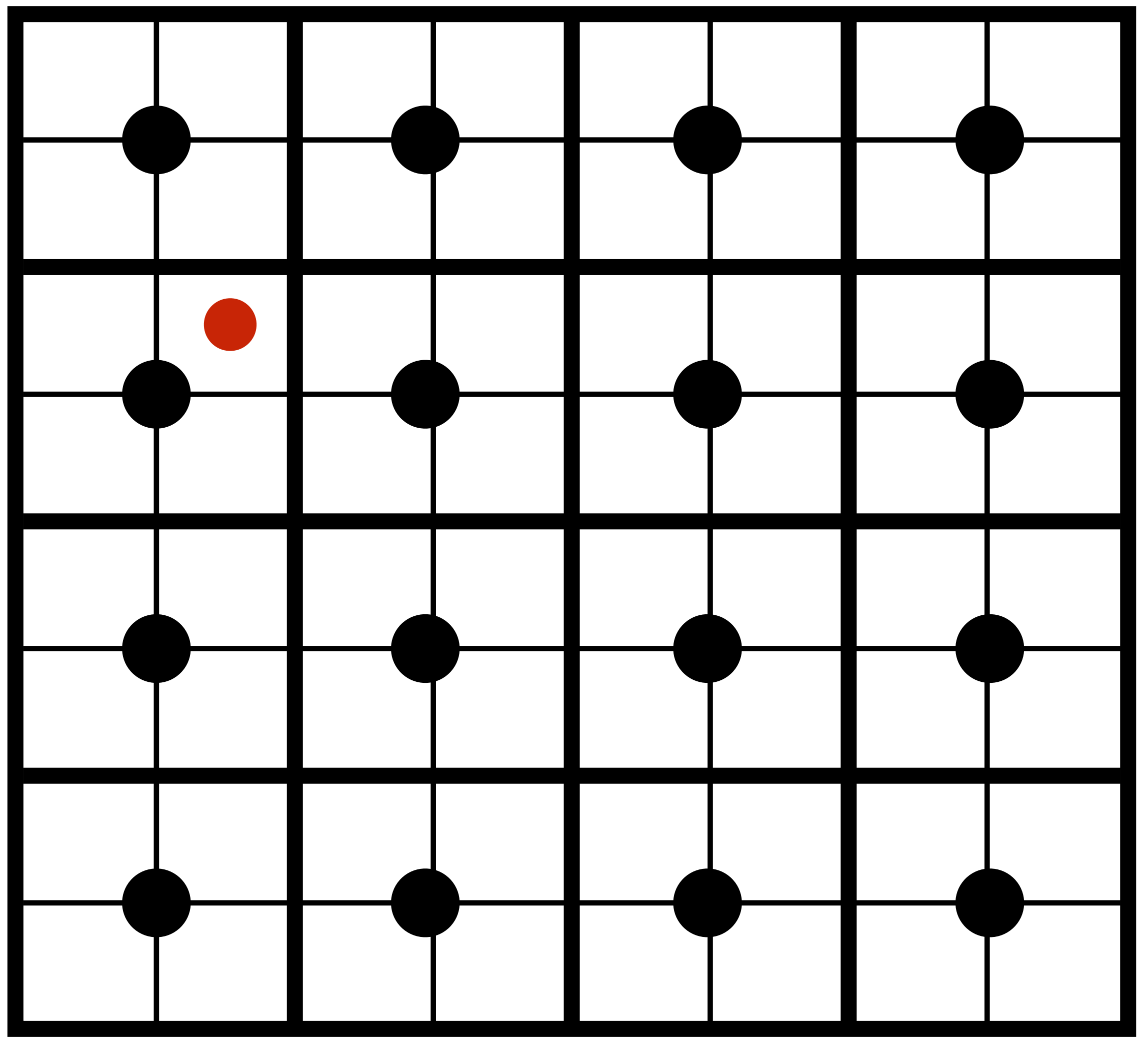
```
float input[(2*WIDTH+2) * (2*HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1/64, 3/64, 3/64, 1/64,    // 4x4 blur (approx Gaussian)
                   3/64, 9/64, 9/64, 3/64,
                   3/64, 9/64, 9/64, 3/64,
                   1/64, 3/64, 3/64, 1/64};

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[(2*j+jj)*(WIDTH+2) + (2*i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH + i] = tmp;
  }
}
```

Upsample

Via bilinear interpolation of samples from low resolution image



```
float input[WIDTH * HEIGHT];
float output[2*WIDTH * 2*HEIGHT];

for (int j=0; j<2*HEIGHT; j++) {
   for (int i=0; i<2*WIDTH; i++) {
     int row = j/2;
      int col = i/2;
      float w1 = (i%2) ? .75 : .25;
      float w2 = (j%2) ? .75 : .25;

      output[j*2*WIDTH + i] = w1 * w2 * input[row*WIDTH + col] +
                              (1.0-w1) * w2 * input[row*WIDTH + col+1] +
                              w1 * (1-w2) * input[(row+1)*WIDTH + col] +
                              (1.0-w1)*(1.0-w2) * input[(row+1)*WIDTH + col+1];
  }
}
```

Gaussian pyramid

$G_0 = \text{image}$

$G_1 = \text{down}(G_0)$

$G_2 = \text{down}(G_1)$

Each image in pyramid contains increasingly low-pass filtered signal

down() = downsample operation

Gaussian pyramid

G_0

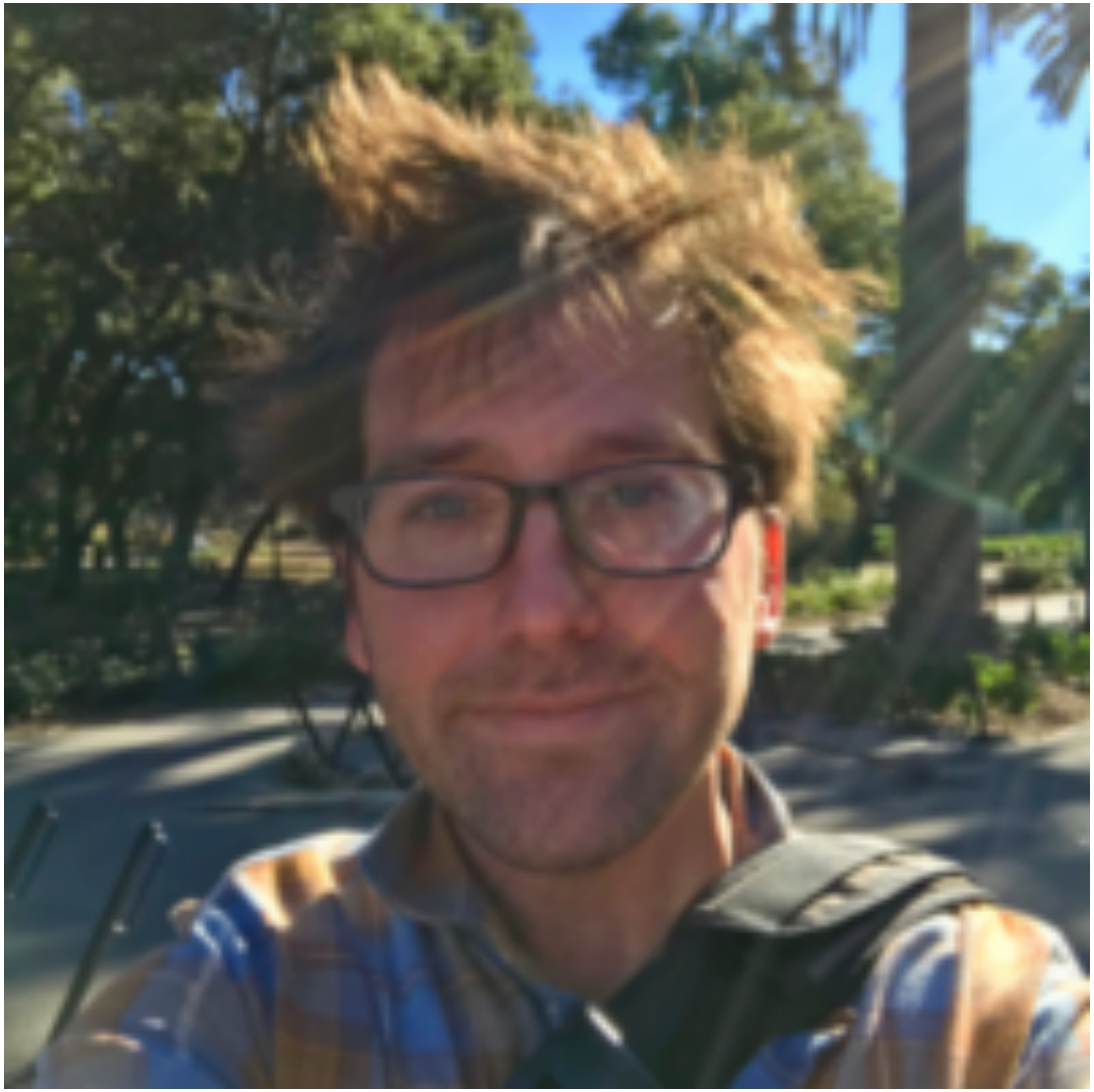
Gaussian pyramid

G_1

Gaussian pyramid

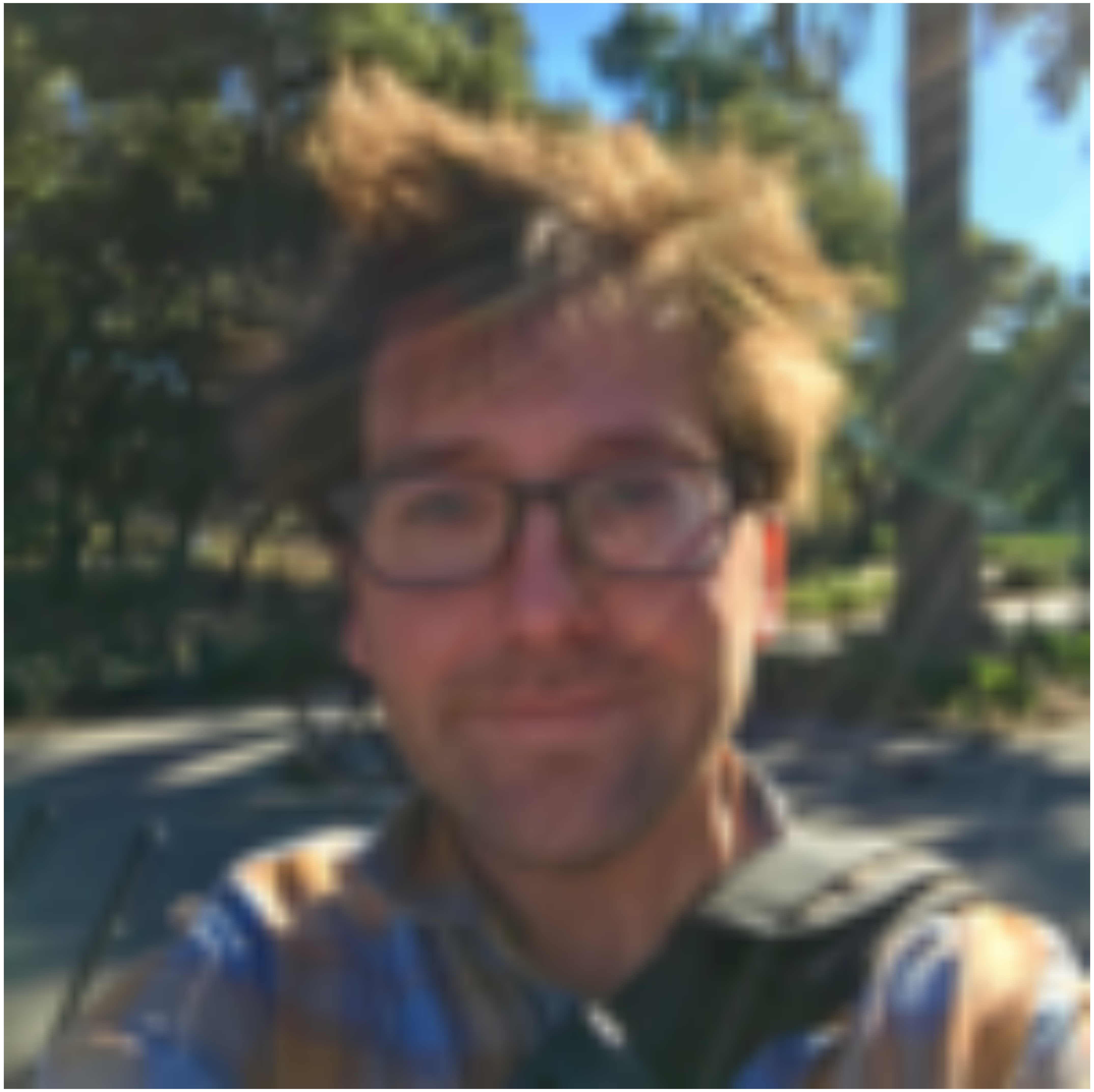
G_2

Gaussian pyramid



G_3

Gaussian pyramid



G_4

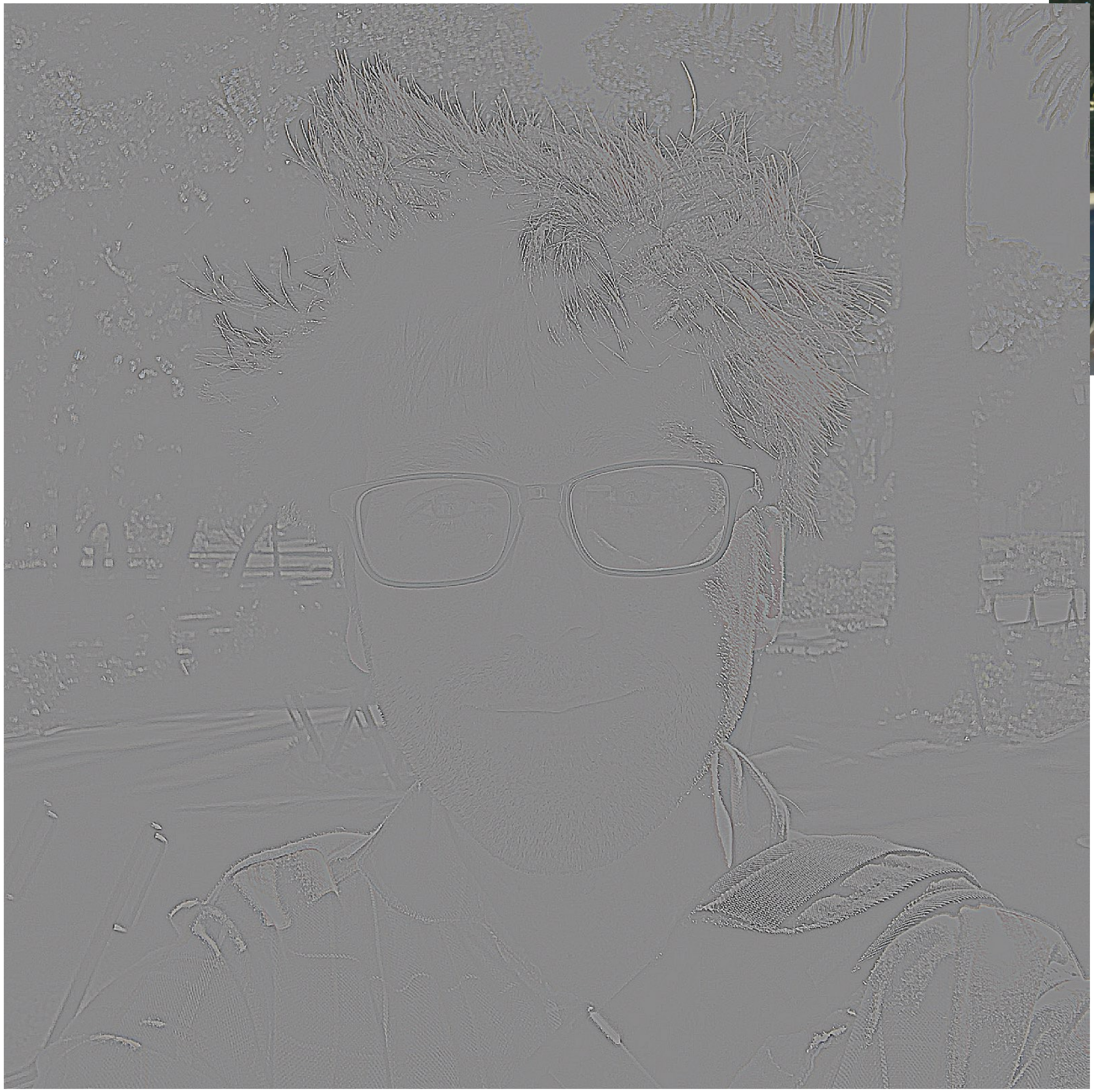
Gaussian pyramid

G_5

Laplacian pyramid

G_0

$G_1 = \text{down}(G_0)$

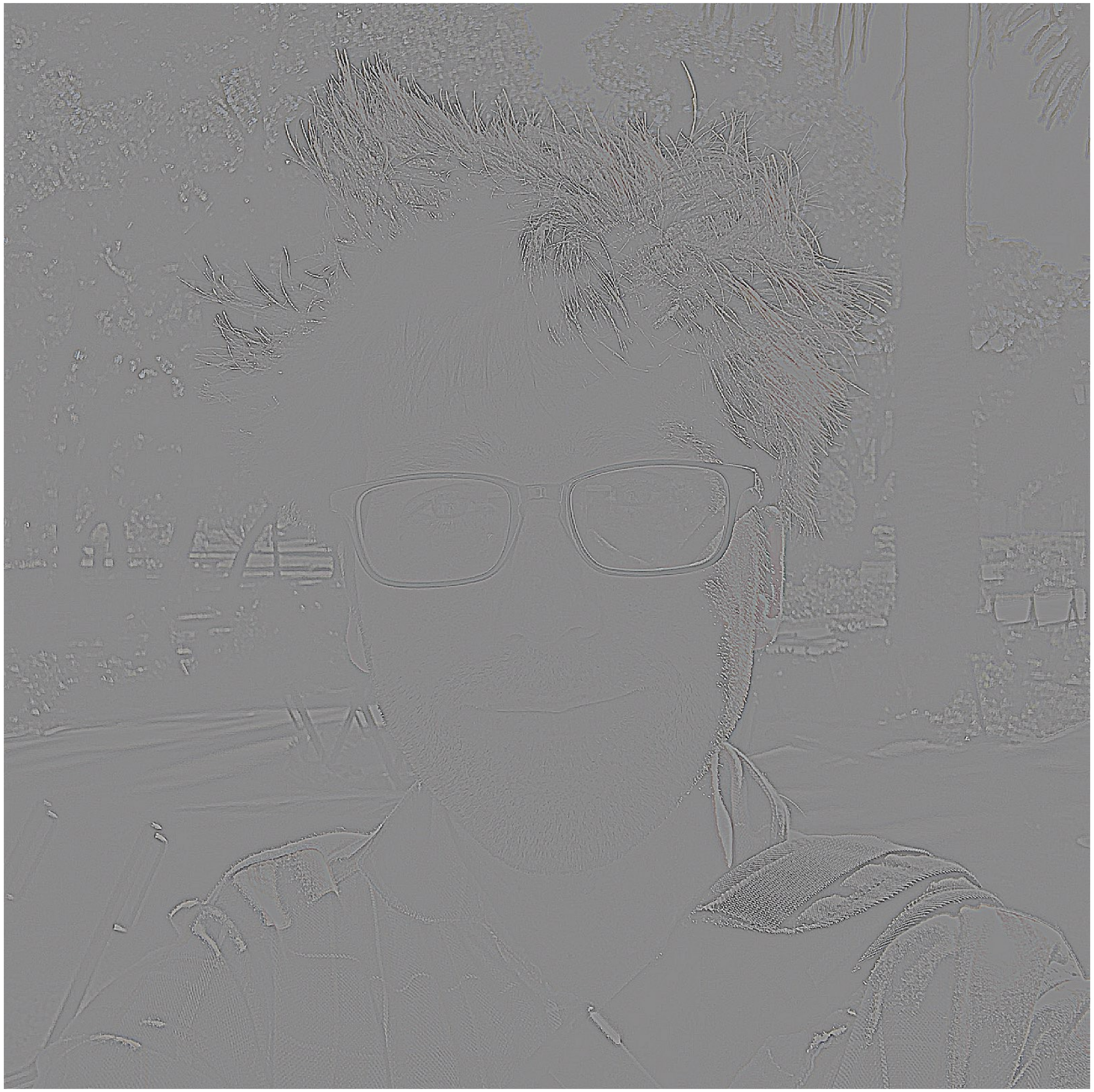


$L_0 = G_0 - \text{up}(G_1)$

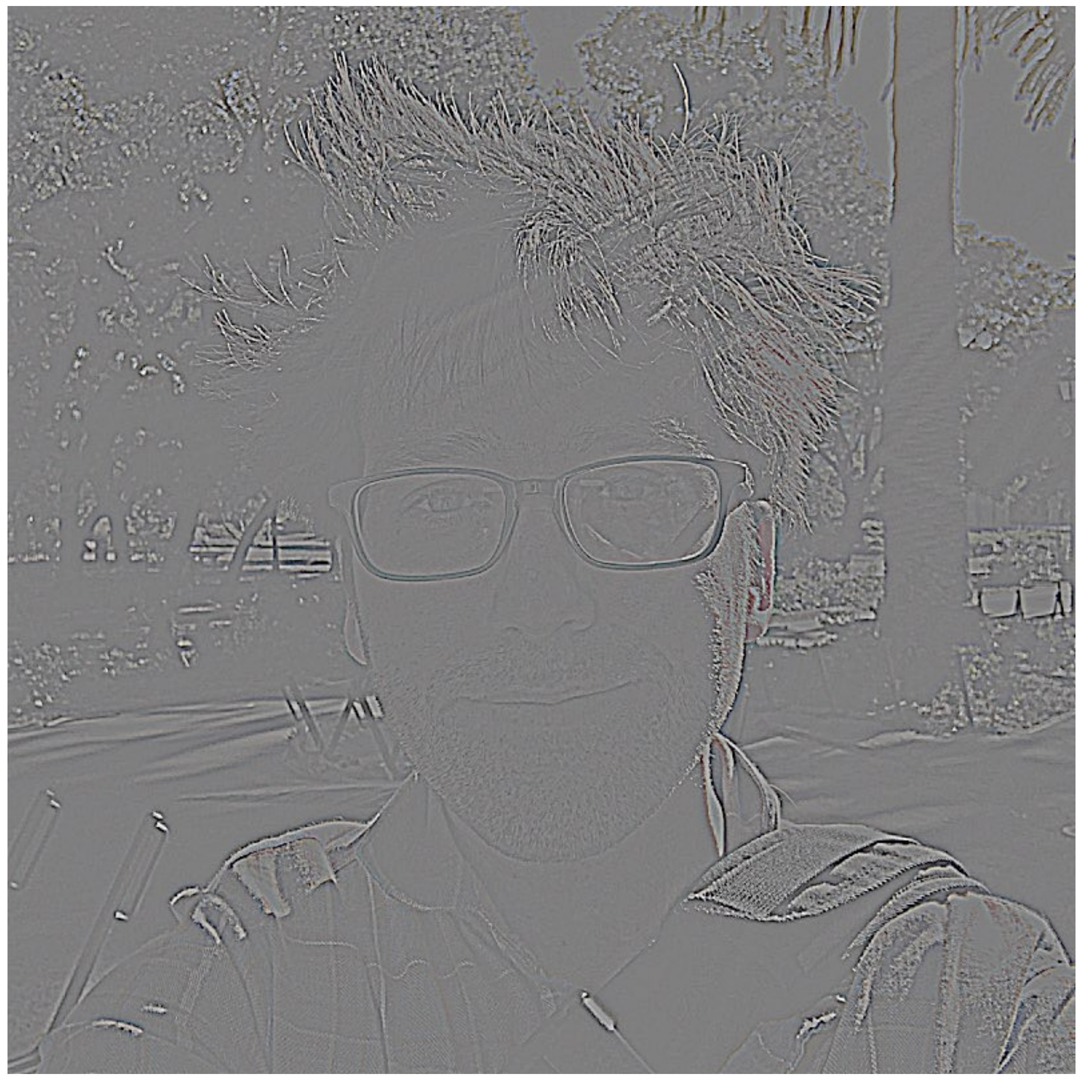
Each image in Laplacian pyramid represents increasingly high frequency content from image

[Burt and Adelson 83]

Laplacian pyramid

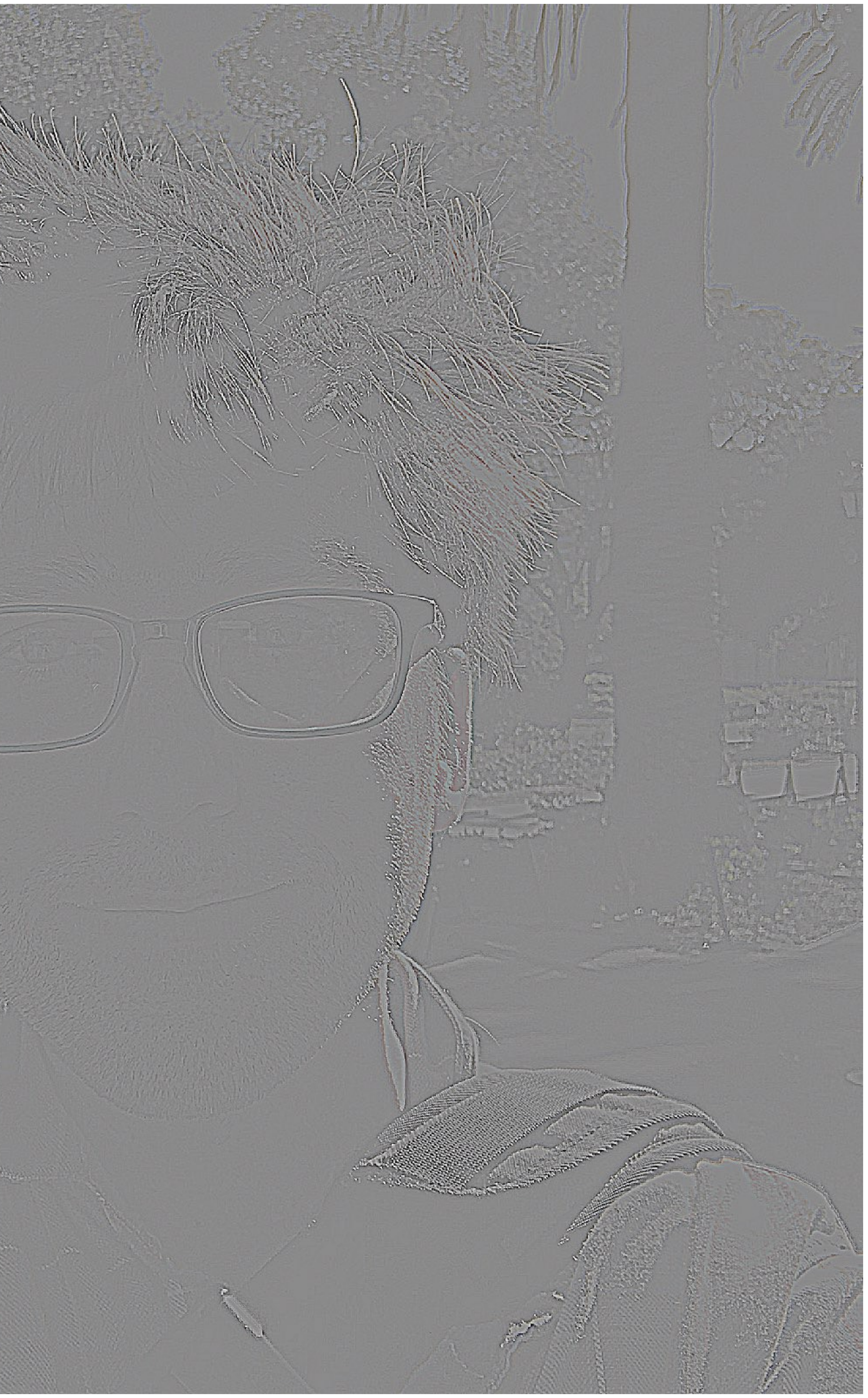


$L_0 = G_0 - up(G_1)$

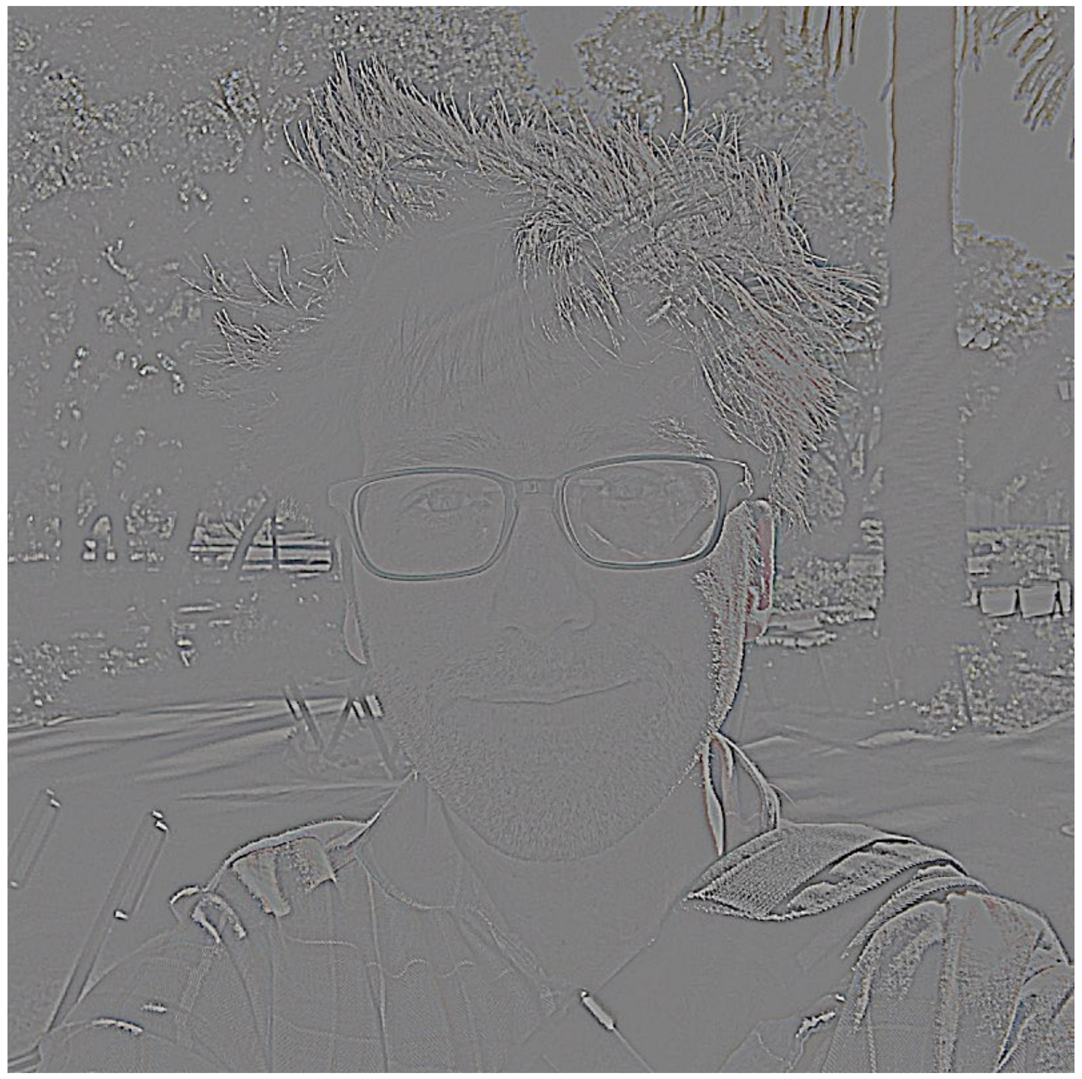


$L_1 = G_1 - up(G_2)$

Laplacian pyramid



$L_0 = G_0 - \text{up}(G_1)$



$L_1 = G_1 - \text{up}(G_2)$

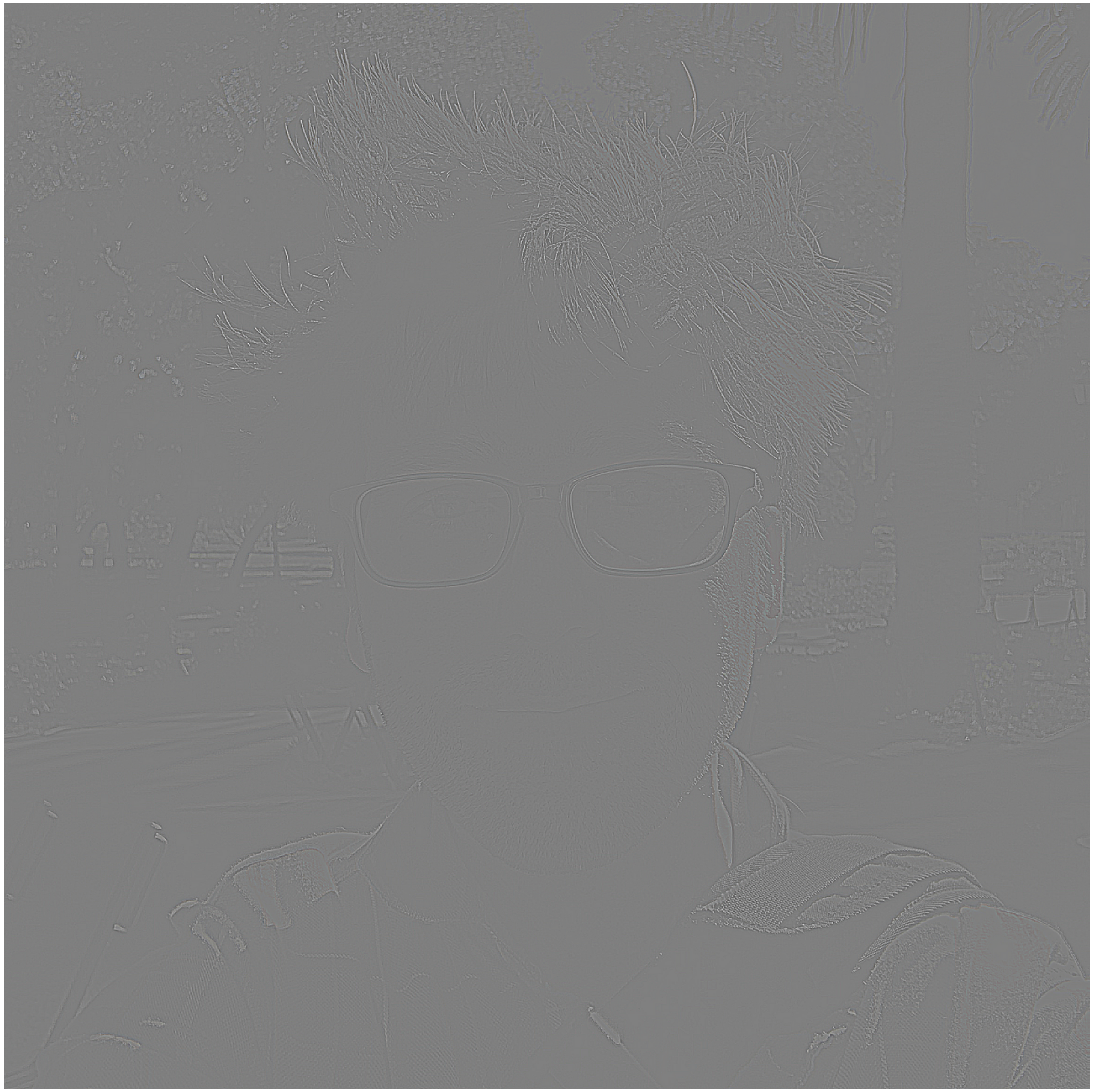
$L_2 = G_2 - \text{up}(G_3)$

$L_3 = G_3 - \text{up}(G_4)$

$L_4 = G_4$

Question: how do you reconstruct original image from its Laplacian pyramid?

Laplacian pyramid

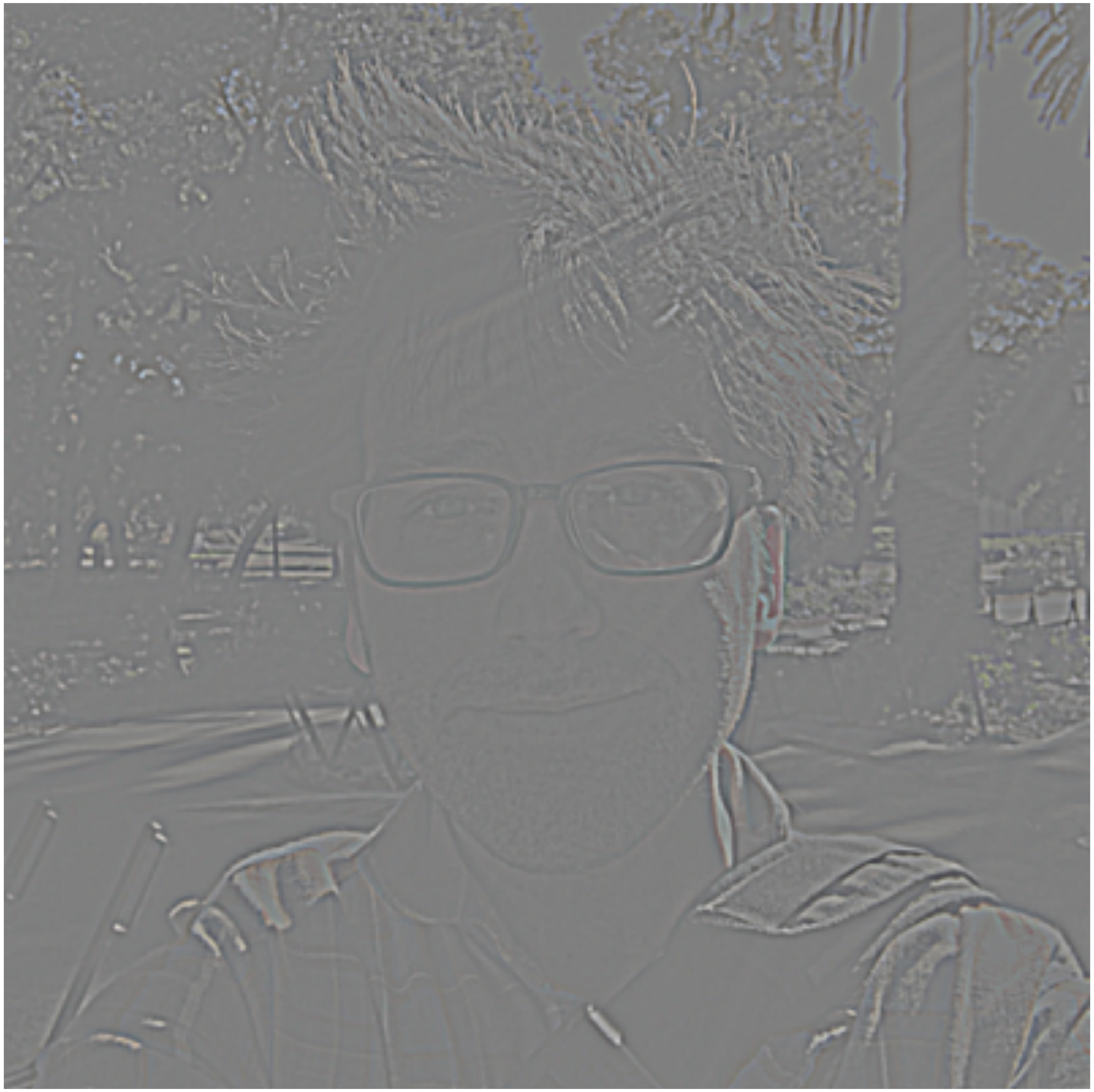


$L_0 = G_0 - up(G_1)$

Laplacian pyramid

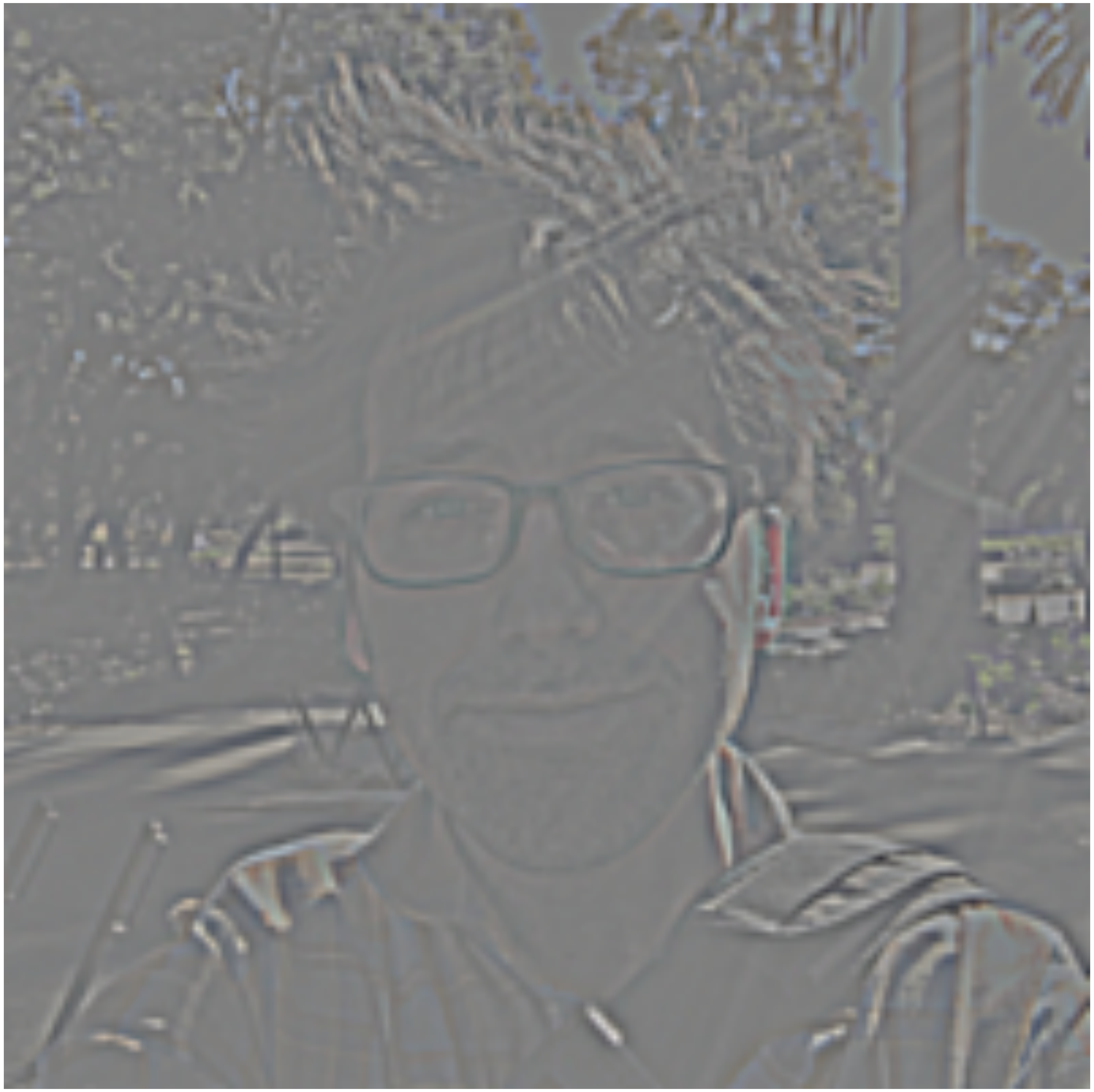
$L_1 = G_1 - \mathrm{up}(G_2)$

Laplacian pyramid



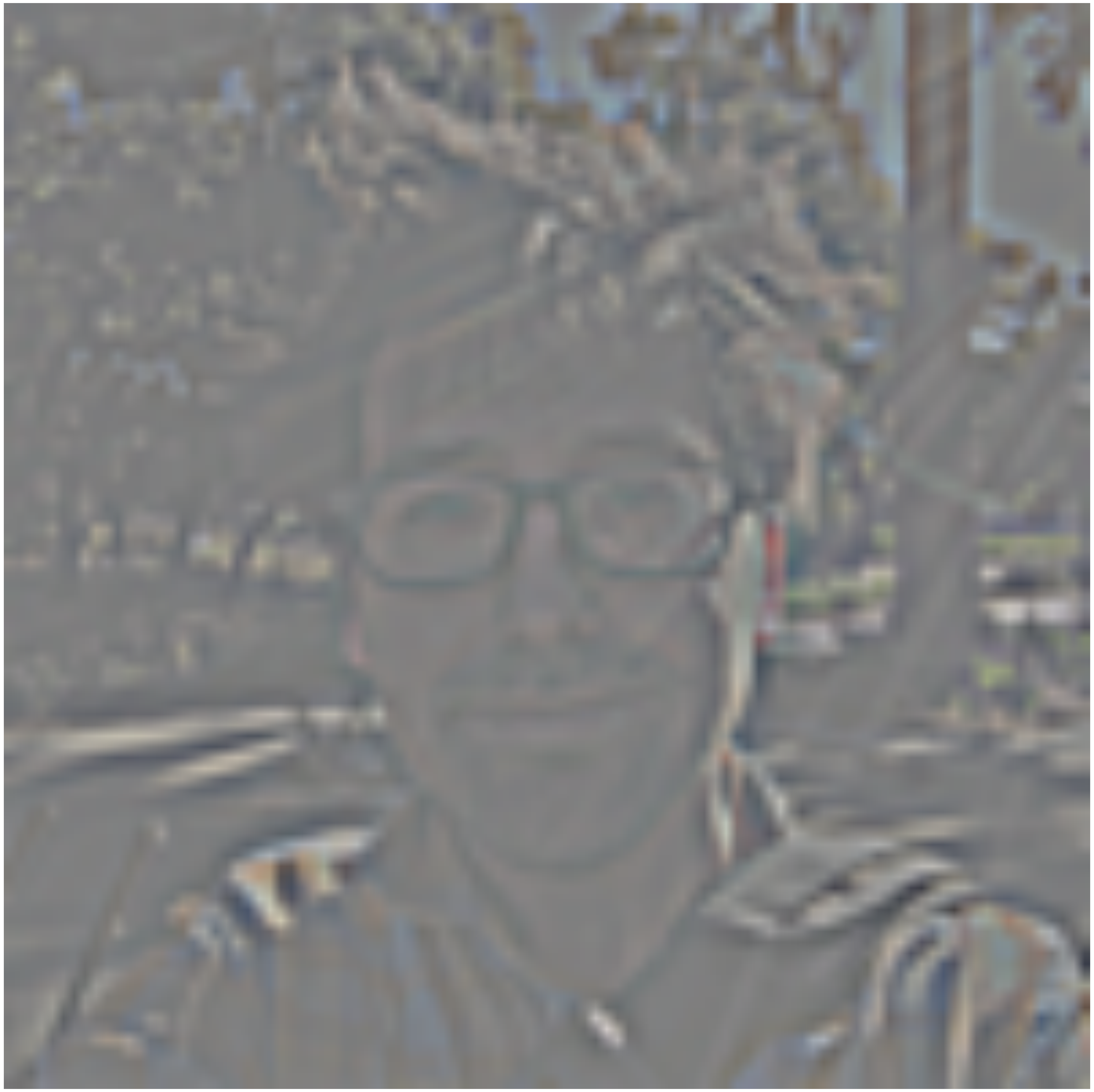
$L_2 = G_2 - \mathrm{up}(G_3)$

Laplacian pyramid



$L_3 = G_3 - \mathrm{up}(G_4)$

Laplacian pyramid



$L_4 = G_4 - \mathrm{up}(G_5)$

Laplacian pyramid

$L_5 = G_5$

Summary

- Gaussian and Laplacian pyramids are image representations where each pixel maintains information about frequency content in a region of the image

- $G_i(x,y)$ — frequencies up to limit given by *i*

- $L_i(x,y)$ — frequencies added to G_{i+1} to get G_i

- Notice: to boost band of frequencies in image around pixel (x,y), increase coefficient $L_i(x,y)$ in Laplacian pyramid

Reducing range of an edge (without reducing detail)

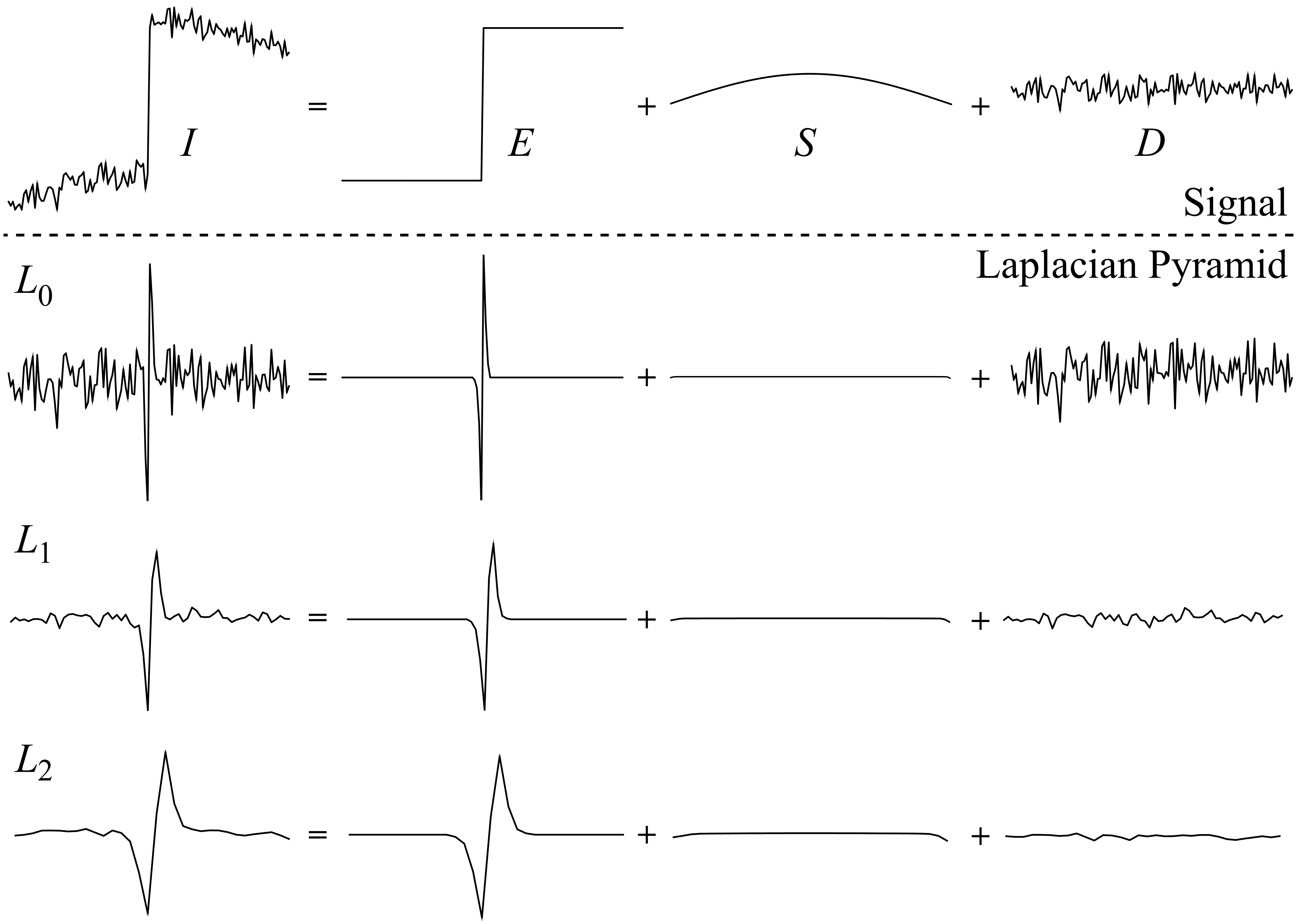
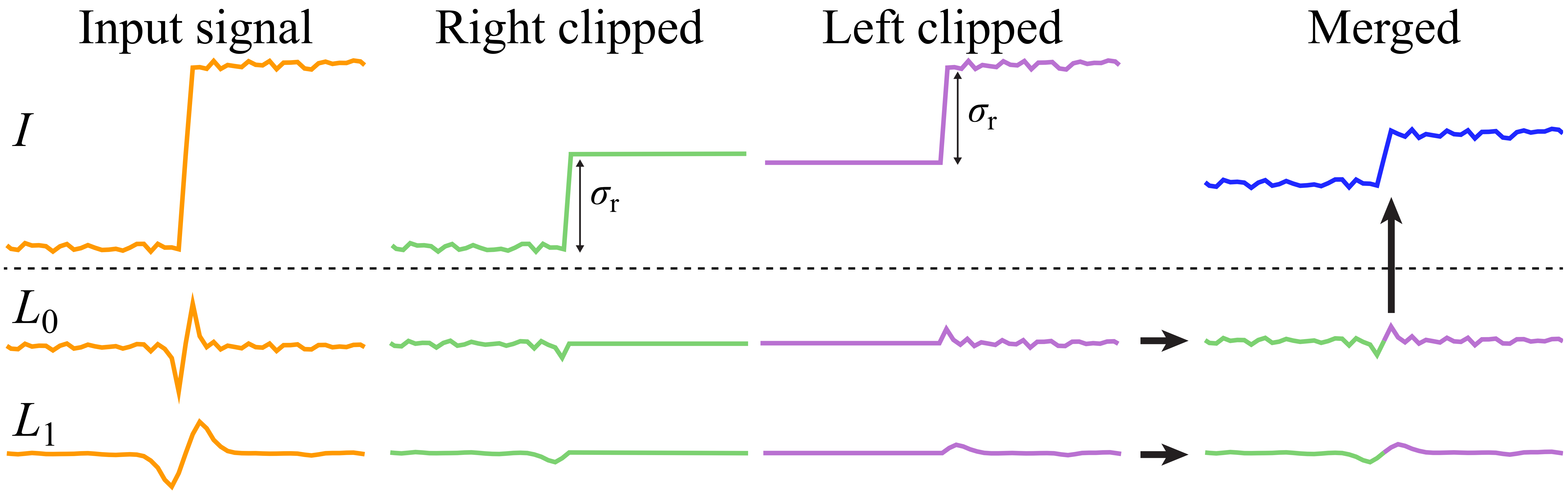


Image credit: Paris et al. 2015

Reducing range of an edge (without reducing detail)



Local Laplacian filtering:

Compute $G_i(x,y)$ — semi-local statistic around (x,y) (less global for larger *i*)

Modify original image I based on $G_i(x,y)$ to get I'=f(I)

Here: f() clamps values that are far from $G_i(x,y)$

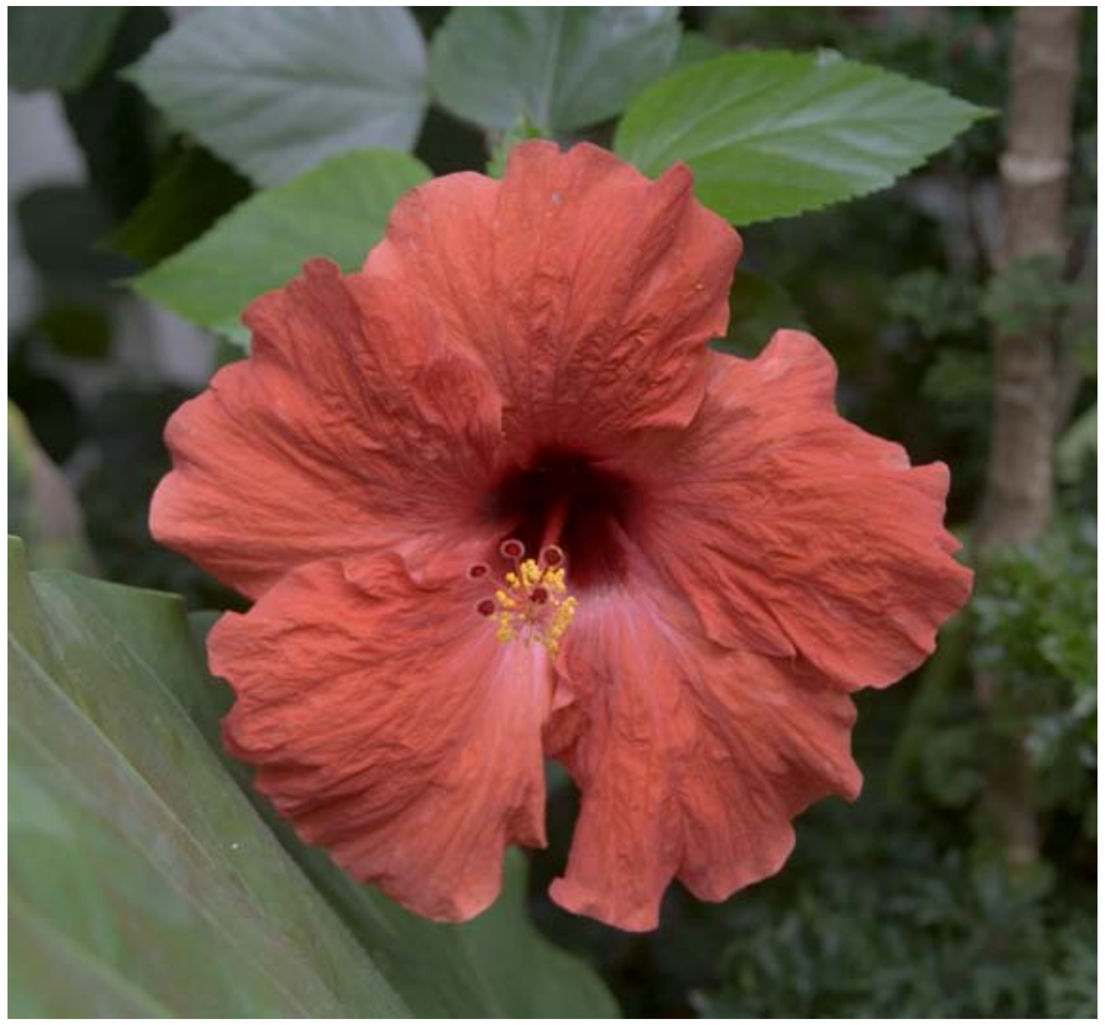
Compute Laplacian pyramid from I' : L(I')

Set coefficient $L_i(x,y)$ of output Laplacian pyramid to $L(I')_i(x,y)$

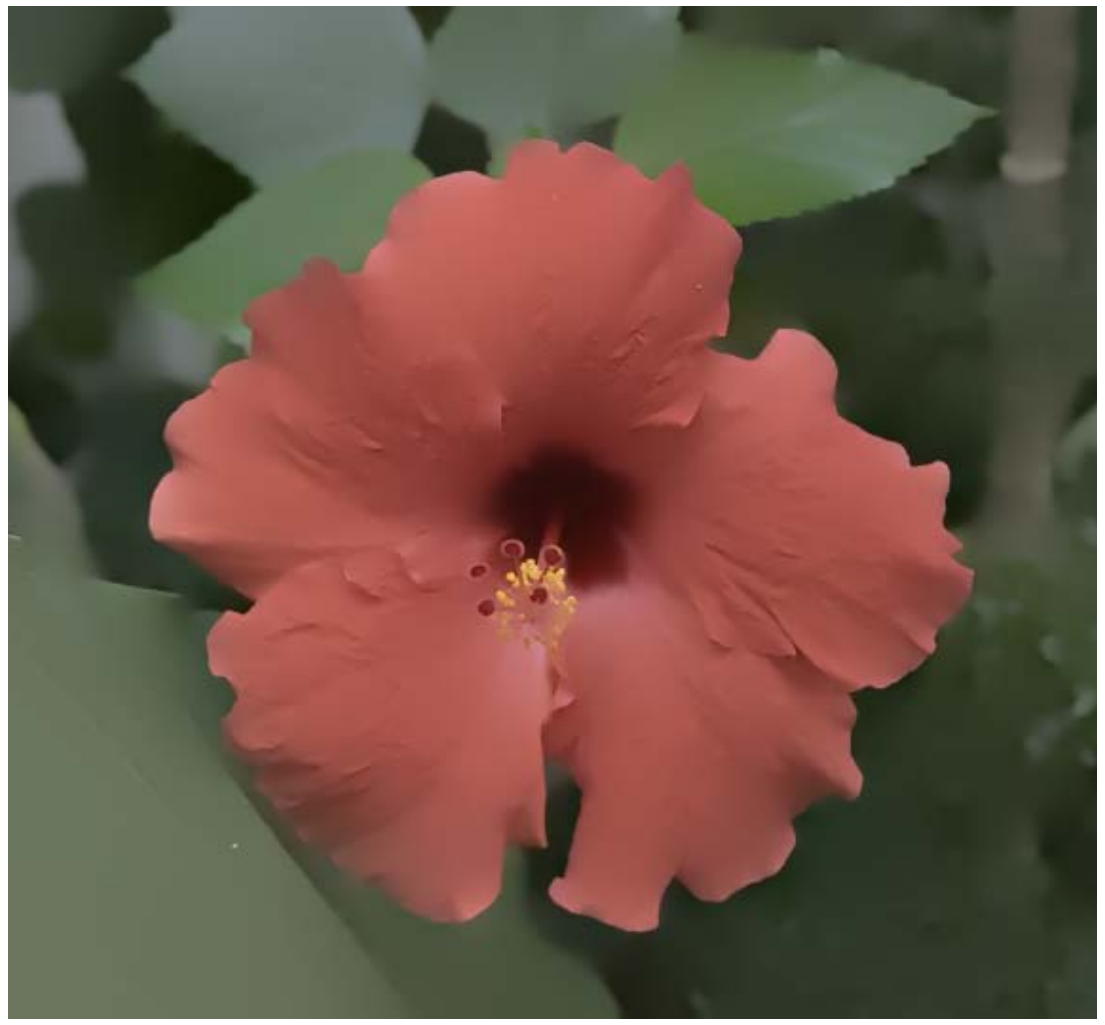
Collapse output Laplacian pyramid to obtain final image.

Image credit: Paris et al. 2015

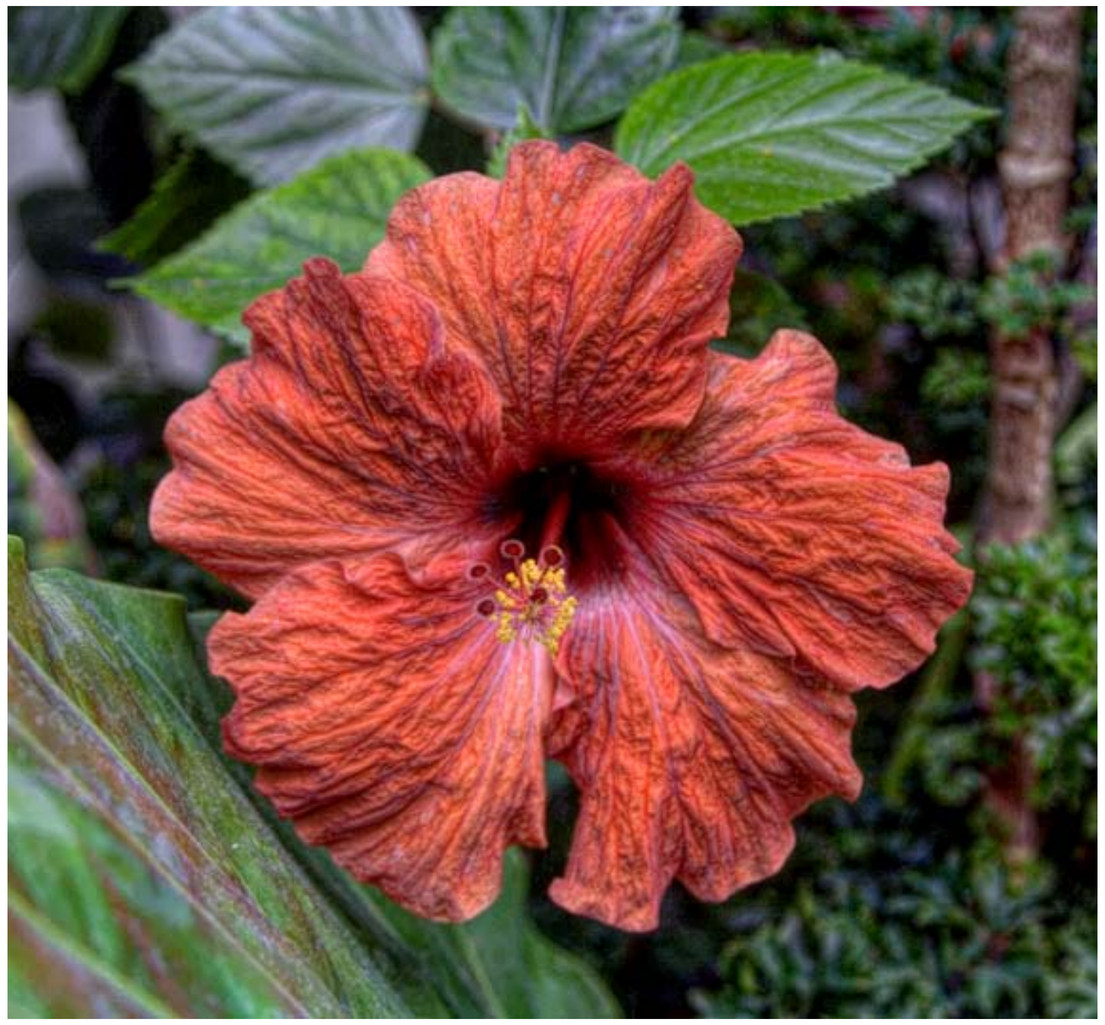
Local laplacian filtering example



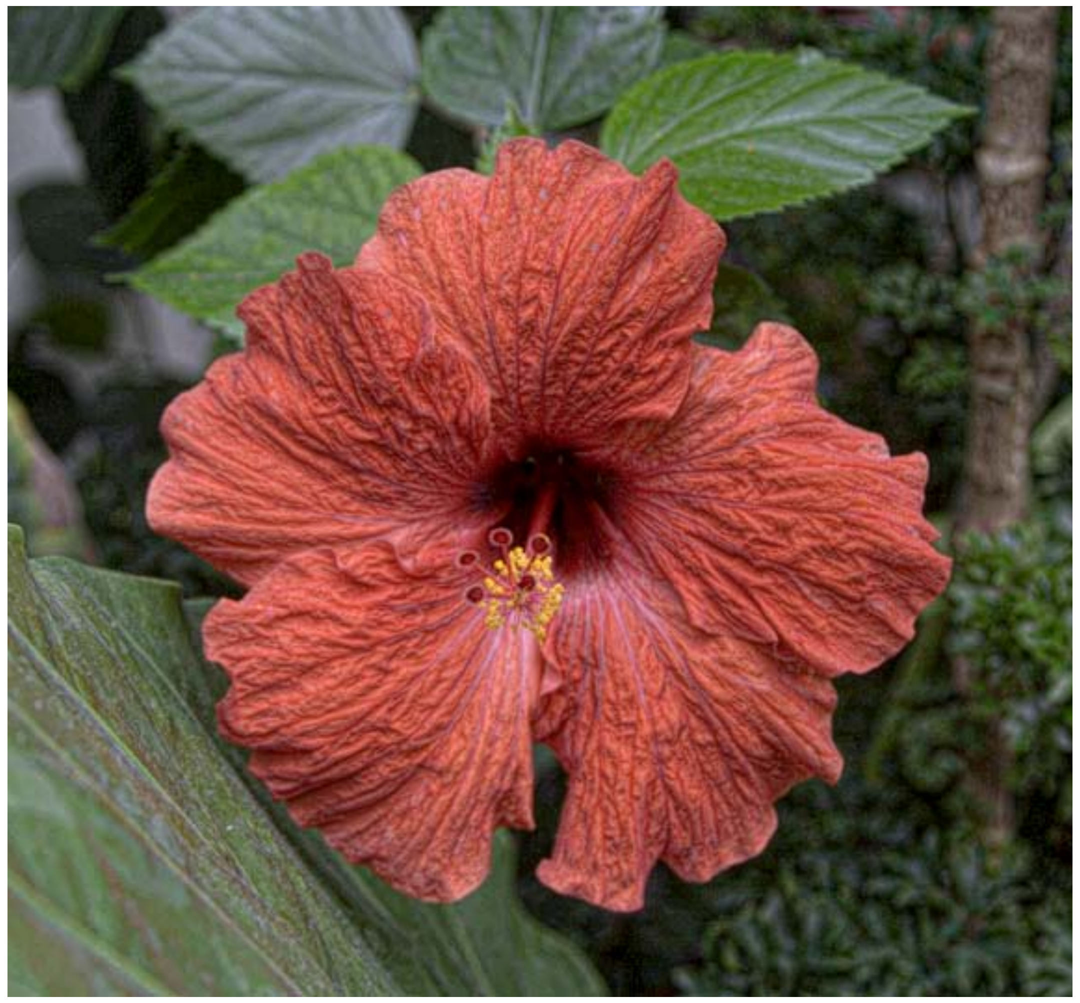
Original Image



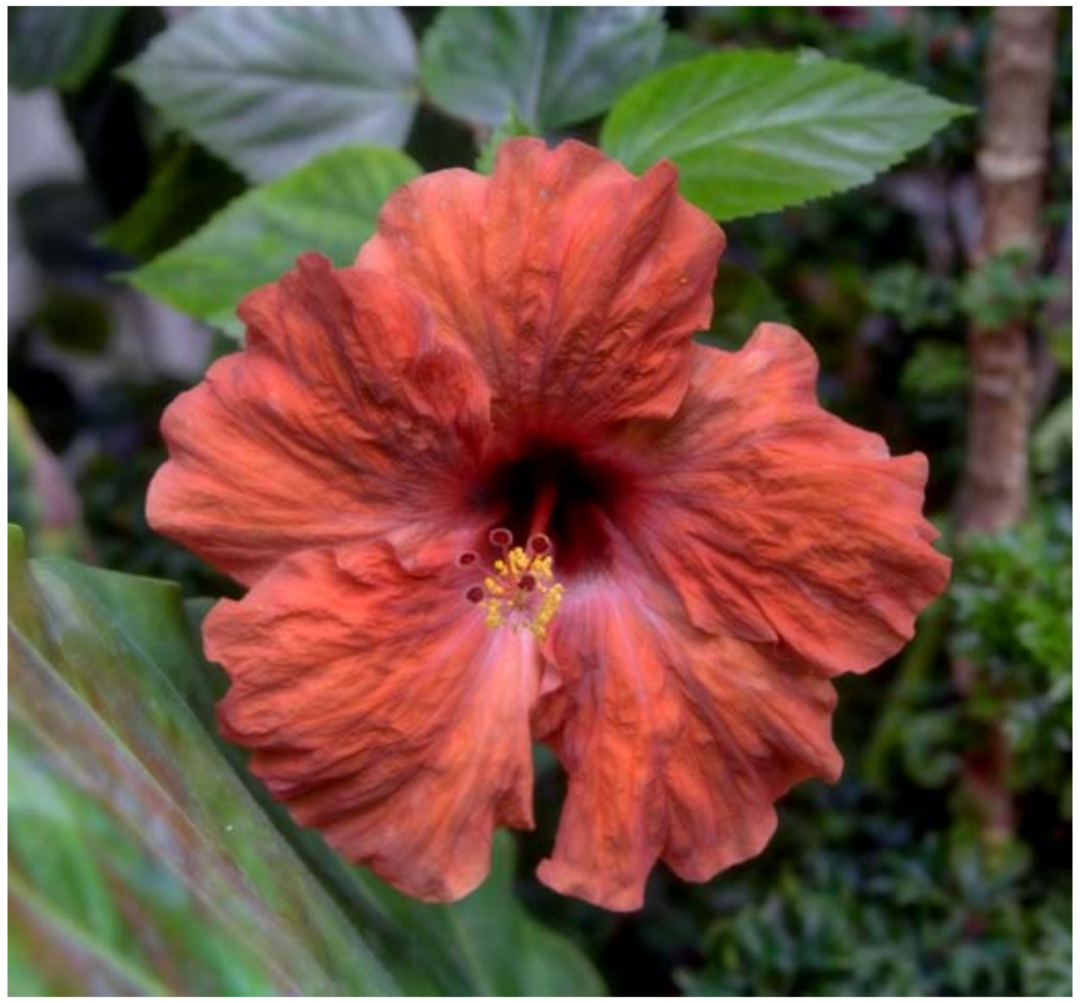
Detail removed



Detail increased (all levels)



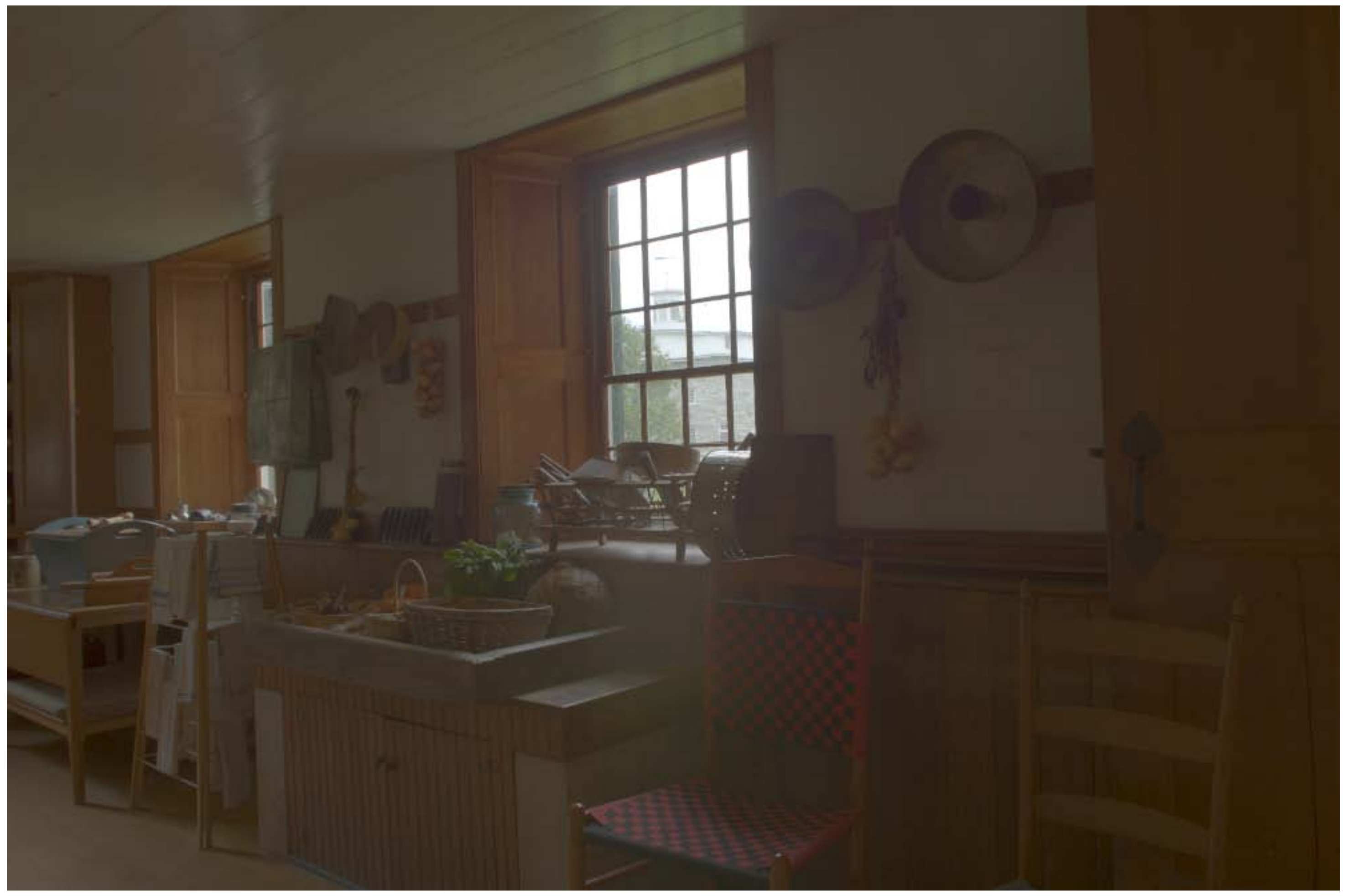
Detail increased (lowest two levels)



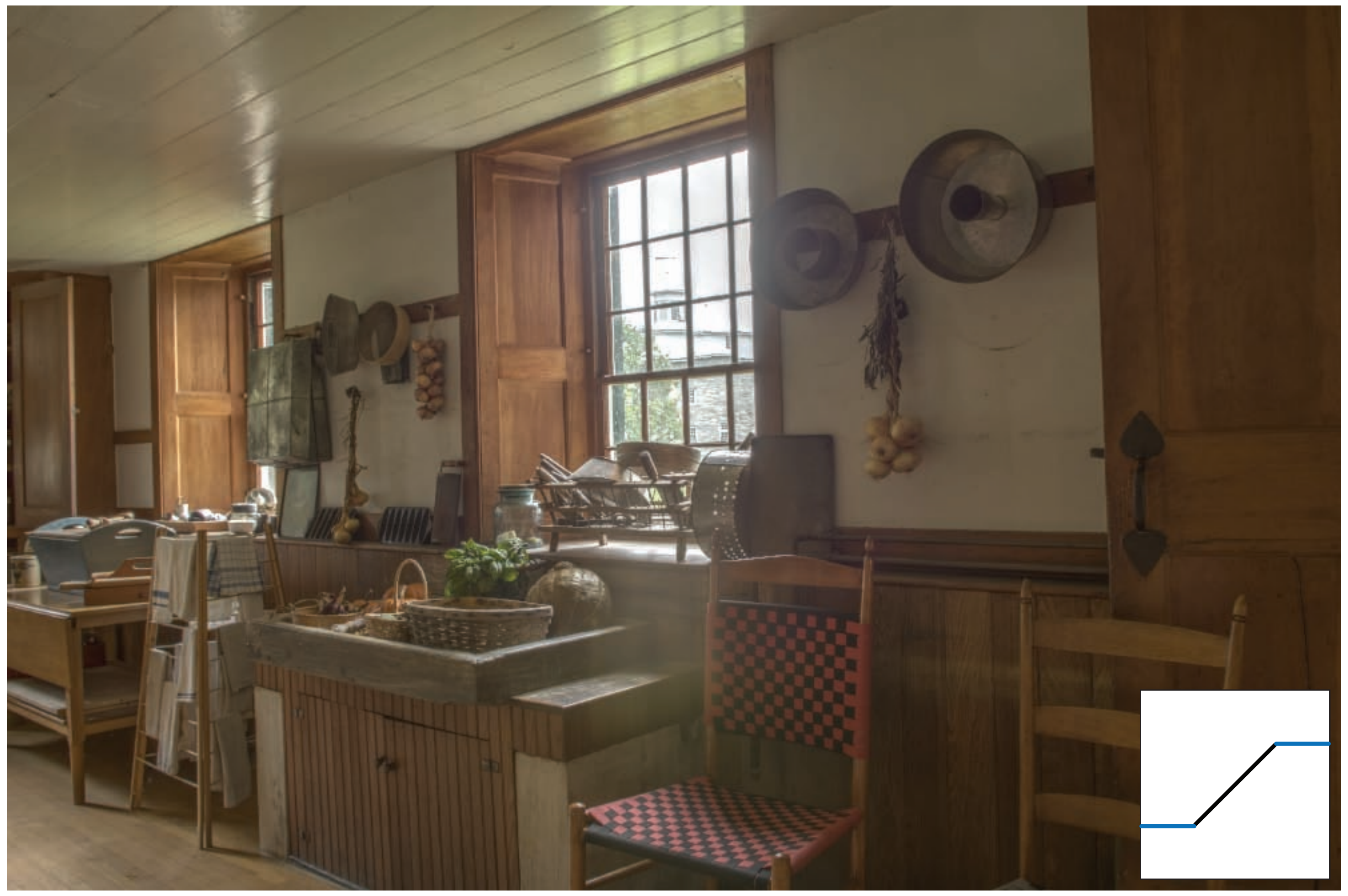
Detail increased (level 3 and higher)

Image credit: Paris et al. 2015

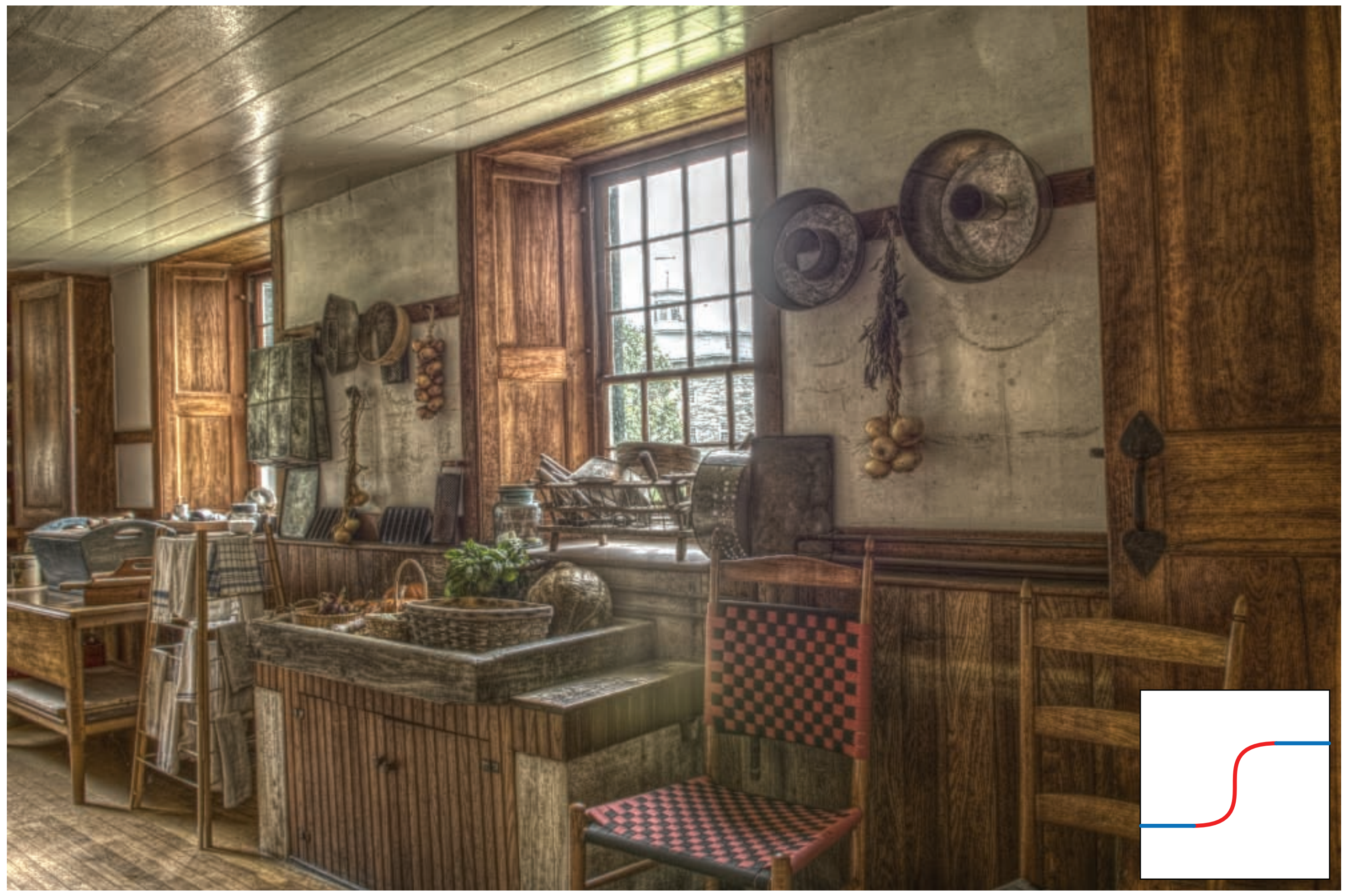
Local laplacian filtering example



(a) input HDR image tone-mapped with a simple gamma curve (details are compressed)



(b) our pyramid-based tone mapping, set to preserve details without increasing them



(c) our pyramid-based tone mapping, set to strongly enhance the contrast of details

Image credit: Paris et al. 2015

Image processing workload characteristics

- “Pointwise operations”:
  - output_pixel = f(input_pixel)

- “Stencil” computations (e.g., convolution, demosaic, etc.)
  - output pixel (x,y) depends on <u>fixed-size</u> local region of input around (x,y)

- Lookup tables

- Multi-resolution operations (upsampling/downsampling)

- Long sequences of operations

Bonus slides:
Estimating motion using optical flow

Optical flow

Goal: determine 2D screen-space velocity of visible objects in image

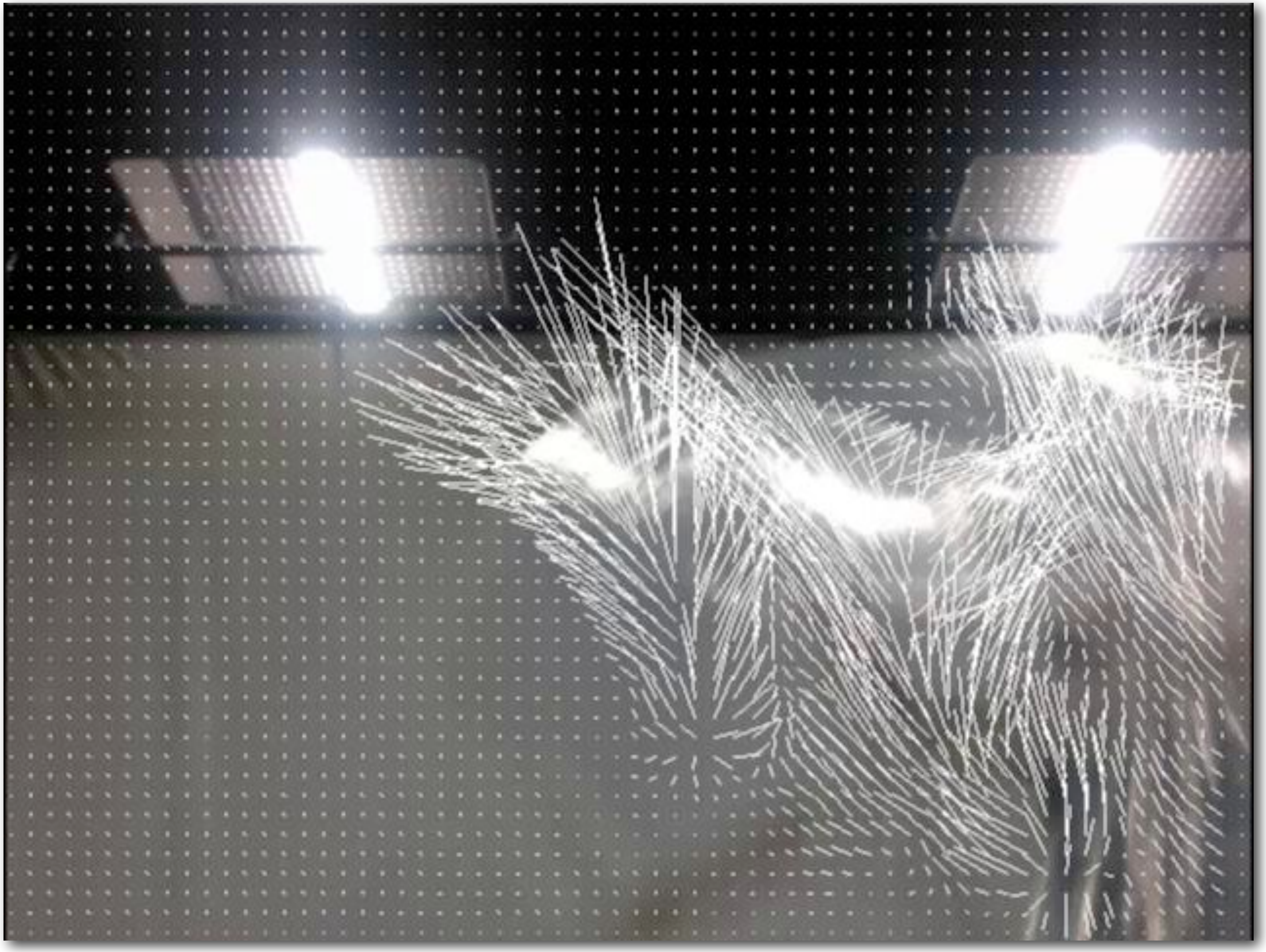


Image source: https://vimeo.com/28395792

Optical flow

- Given image A (at time *t*) and image B (at time *t* + Δ*t*) compute the per-pixel motion needed to correspond the two images
- Major assumption 1: "brightness constancy"
  - The appearance of a scene surface point that is visible in both images A and B is the same in both images

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t)$$

The point observed at (*x*,*y*) at time *t* moves to (*x*+Δx, *y*+Δy) at *t*+Δ*t*
*(*and has the same appearance in both situations)

Taylor expansion

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t + \text{higher order terms}$$

So...

$$I(x, y, t) \approx I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t$$

$$\boxed{I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y} + \boxed{I_t(x, y, t)\Delta t} = 0$$

The observed change in pixel (x,y)

Is due to object motion at point by (Δx, Δy)

AP Photo/Daniel Ochoa De Olza

Gradient-constraint equation for a pixel is underconstrained

Gradient-constraint equation is insufficient to solve for motion

One equation, two unknowns: (Δx, Δy)

$$\boxed{I_x(x, y, t)}\Delta x + \boxed{I_y(x, y, t)}\Delta y + \boxed{I_t(x, y, t)\Delta t} = 0$$

Known: observed change in pixel (x,y) over consecutive frames

Known: spatial image gradients in image A

Major assumption 2: nearby pixels have similar motion (Lucas-Kanade)

$$I_x(x_0, y_0, t)\Delta x + I_y(x_0, y_0, t)\Delta y + I_t(x_0, y_0, t)\Delta t = 0$$

$$I_x(x_1, y_1, t)\Delta x + I_y(x_1, y_1, t)\Delta y + I_t(x_1, y_1, t)\Delta t = 0$$

$$I_x(x_2, y_2, t)\Delta x + I_y(x_2, y_2, t)\Delta y + I_t(x_2, y_2, t)\Delta t = 0$$

$$\vdots$$

Now we have a overconstrained system, compute least squares solution

Weighted least-squares solution

Gradient-constraint equation is insufficient to solve for motion

One equation, two unknowns: (Δx, Δy)

$$I_x(x_0, y_0, t)\Delta x + I_y(x_0, y_0, t)\Delta y + I_t(x_0, y_0, t)\Delta t = 0$$

$$I_x(x_1, y_1, t)\Delta x + I_y(x_1, y_1, t)\Delta y + I_t(x_1, y_1, t)\Delta t = 0$$

$$I_x(x_2, y_2, t)\Delta x + I_y(x_2, y_2, t)\Delta y + I_t(x_2, y_2, t)\Delta t = 0$$

$$\vdots$$

Compute weighted least squares solution by minimizing:

$$E(\Delta x, \Delta y) = \sum_{x_i, y_i} w(x_i, y_i, x, y)\left[I_x(x_i, y_i, t)\Delta x + I_y(x_i, y_i, t)\Delta y + I_t(x_i, y_i, t)\Delta t\right]^2$$

(*x*$_i$, *y*$_i$) are pixels in region around (*x*,*y*).

Weighting function *w*() weights error contribution based on distance between (*x*$_i$, *y*$_i$) and (*x*, *y*) e.g., Gaussian fall-off.

Solving for motion

E (Δx, Δy) minimized when derivatives are zero:

$$\frac{dE(\Delta x,\Delta y)}{d(\Delta x)} = \sum_{x_i,y_i} w(x_i,y_i,x,y)\left[I_x^2\Delta x + I_xI_y\Delta y + I_xI_t\right] = 0$$

$$\frac{dE(\Delta x,\Delta y)}{d(\Delta y)} = \sum_{x_i,y_i} w(x_i,y_i,x,y)\left[I_y^2\Delta y + I_xI_y\Delta x + I_yI_t\right] = 0$$

Rewrite, now solve the following linear system for Δx, Δy:

$$\Delta x \overset{A0}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_x^2}} + \Delta y \overset{B0}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_y}} + \overset{C0}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_t}} = 0$$

$$\Delta x \overset{A1}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_xI_y}} + \Delta y \overset{B1}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_y^2}} + \overset{C1}{\boxed{\sum_{x_i,y_i} w(x_i,y_i,x,y)I_yI_t}} = 0$$

Precompute partial derivatives I_x, I_y, I_t from original images A and B

For each pixel (*x*,*y*): evaluate A0, B0, C0, A1, B1, C1, then solve for (*Δx*, *Δy*) at (*x*,*y*)

Optical flow, implemented in practice

Gradient-constraint equation makes a linear motion assumption

$$I(x, y, t) \approx I(x, y, t) + I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y + I_t(x, y, t)\Delta t$$

$$\boxed{I_x(x, y, t)\Delta x + I_y(x, y, t)\Delta y} + \boxed{I_t(x, y, t)\Delta t} = 0$$

The observed change in pixel (x,y)

Is due to object motion at point by (Δx, Δy)

- Improvement: iterative techniques use this original flow field to compute higher order residuals (to estimate non-linear motion)

- Question: why is it important for optical flow implementation to be very efficient?
  - Hint: consider linear-motion assumption