Lecture 2:

The Camera Image Processing Pipeline

Visual Computing Systems
CMU 15-769, Fall 2016

Acknowledgements to Ren Ng, Marc Levoy for various slides used in this lecture.

Next two lectures

The values of pixels in photograph you see on screen are quite different than the values output by the photosensor 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

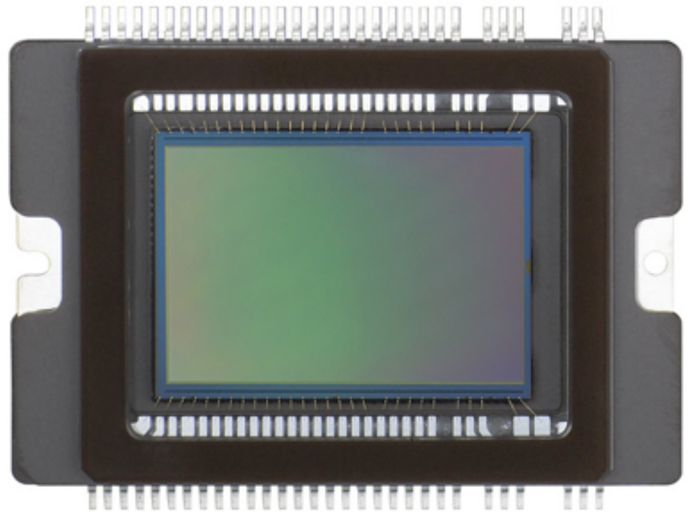
Where we are headed

- I'm about to describe the pipeline of operations that take raw image pixels from a sensor (measurements) to high-quality images
  - 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
  - Consider what a future image analysis pipeline might feature: person identification, action recognition, scene understanding (to automatically compose shot or automatically pick best picture) etc.

Camera cross section

Sensor

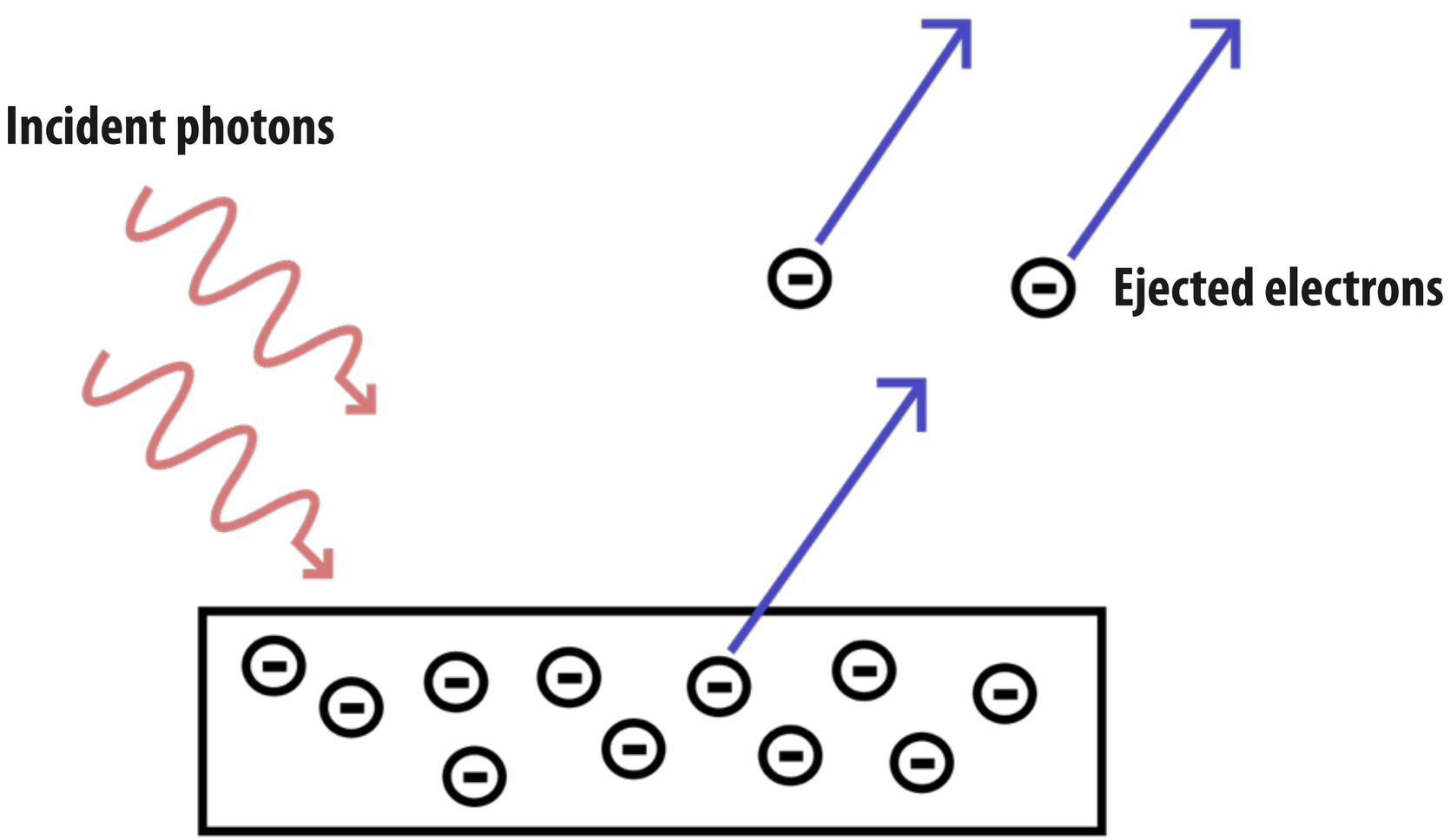


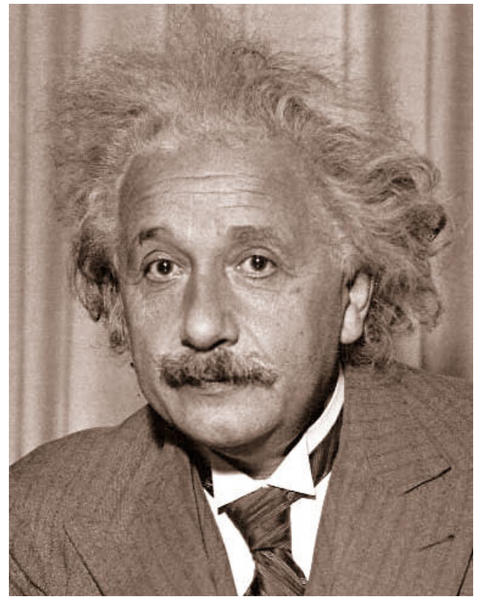
Canon 14 MP CMOS Sensor
(14 bits per pixel)

Image credit: Canon (EOS M)

The Sensor

Photoelectric effect



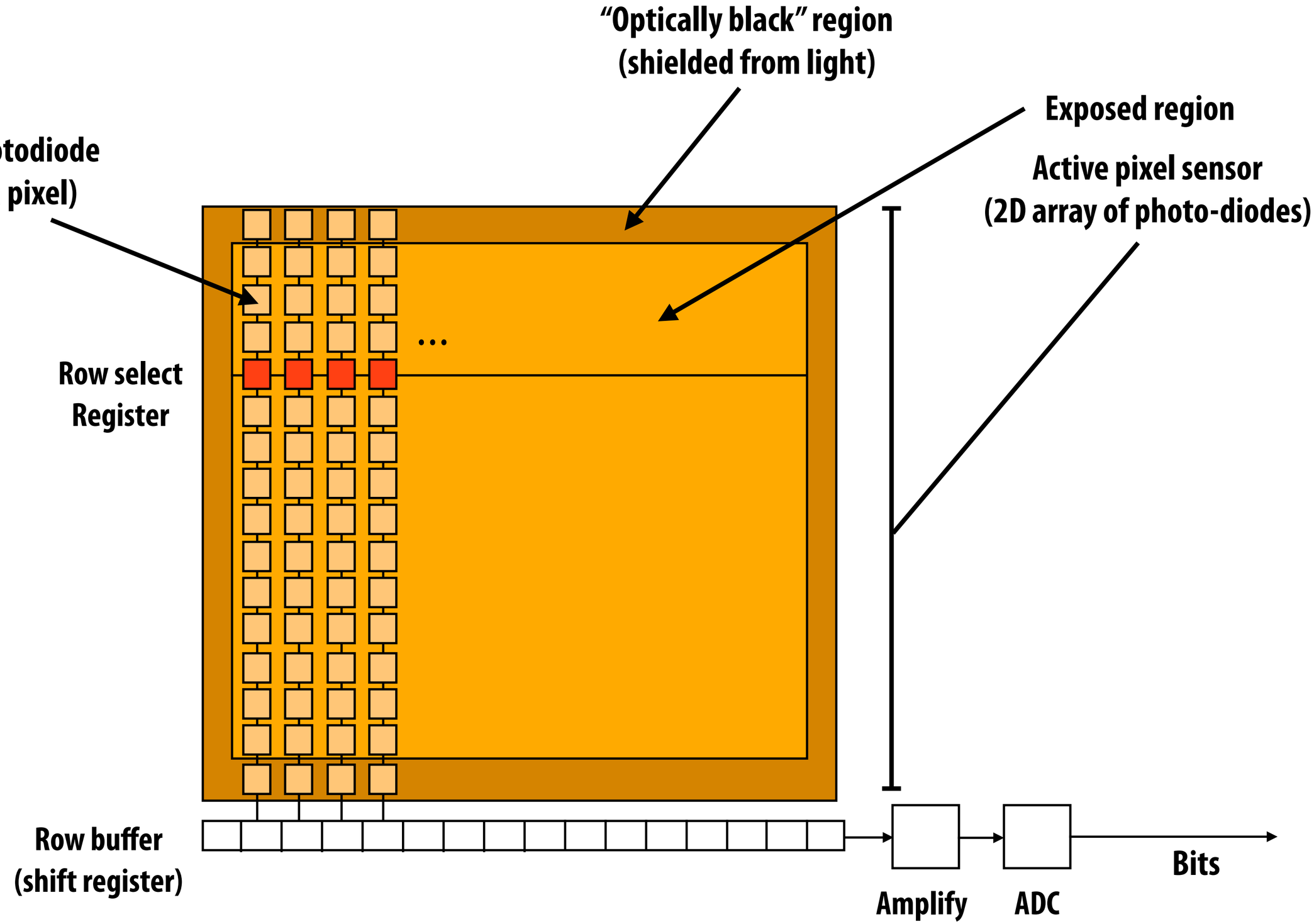


Albert Einstein

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

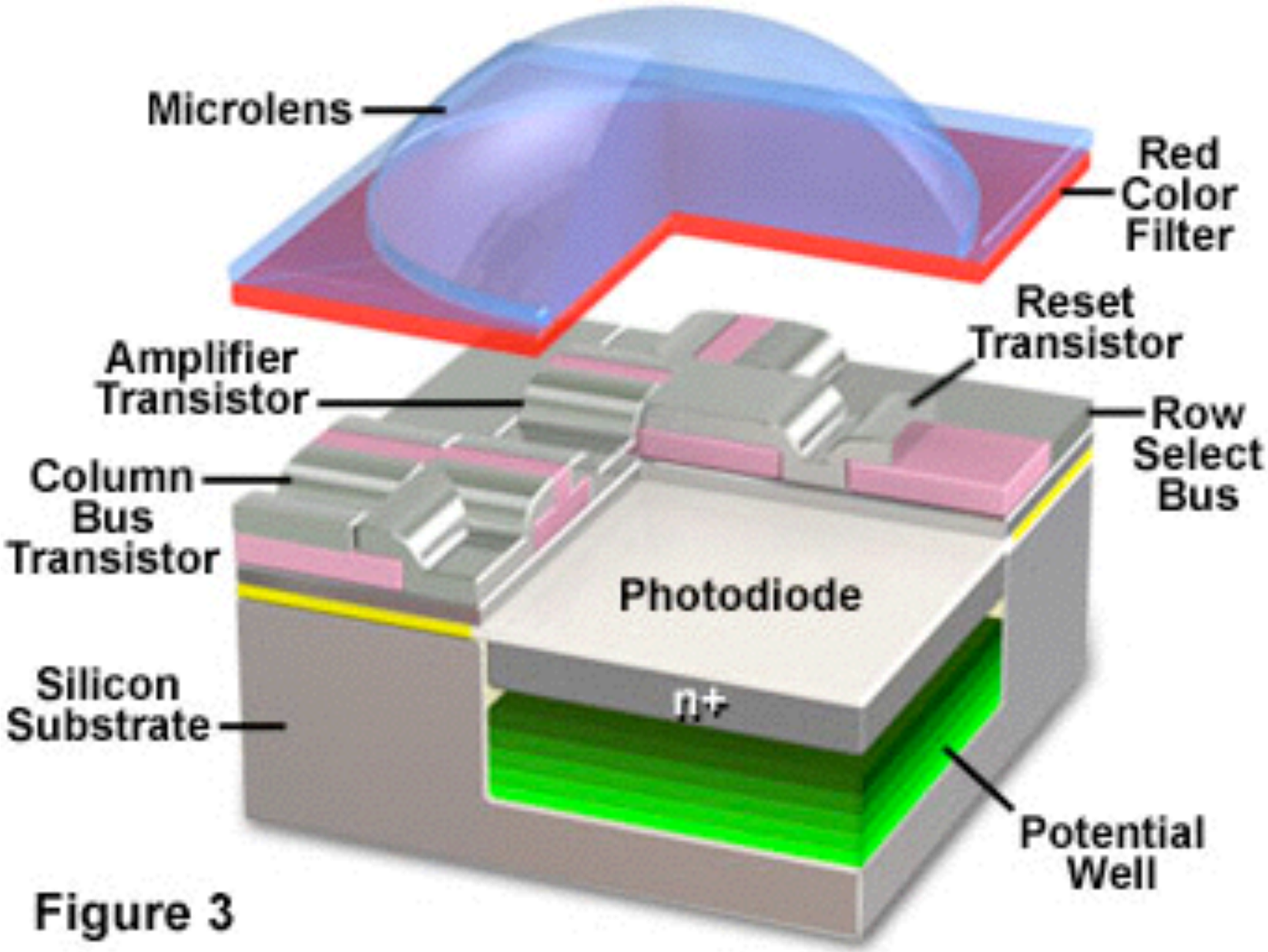
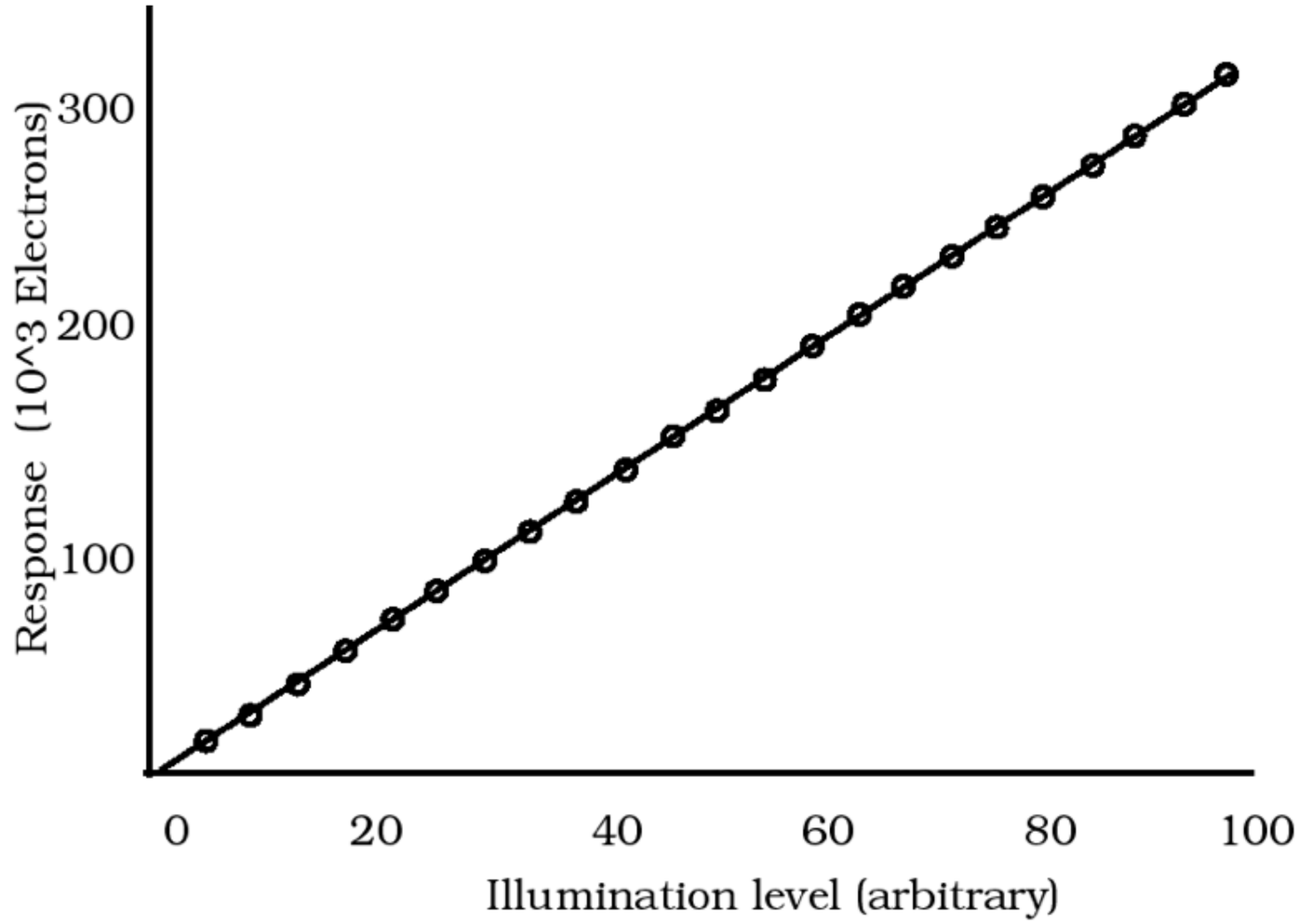


Illustration credit: Molecular Expressions (http://micro.magnet.fsu.edu/primer/digitalimaging/cmosimagesensors.html)

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:

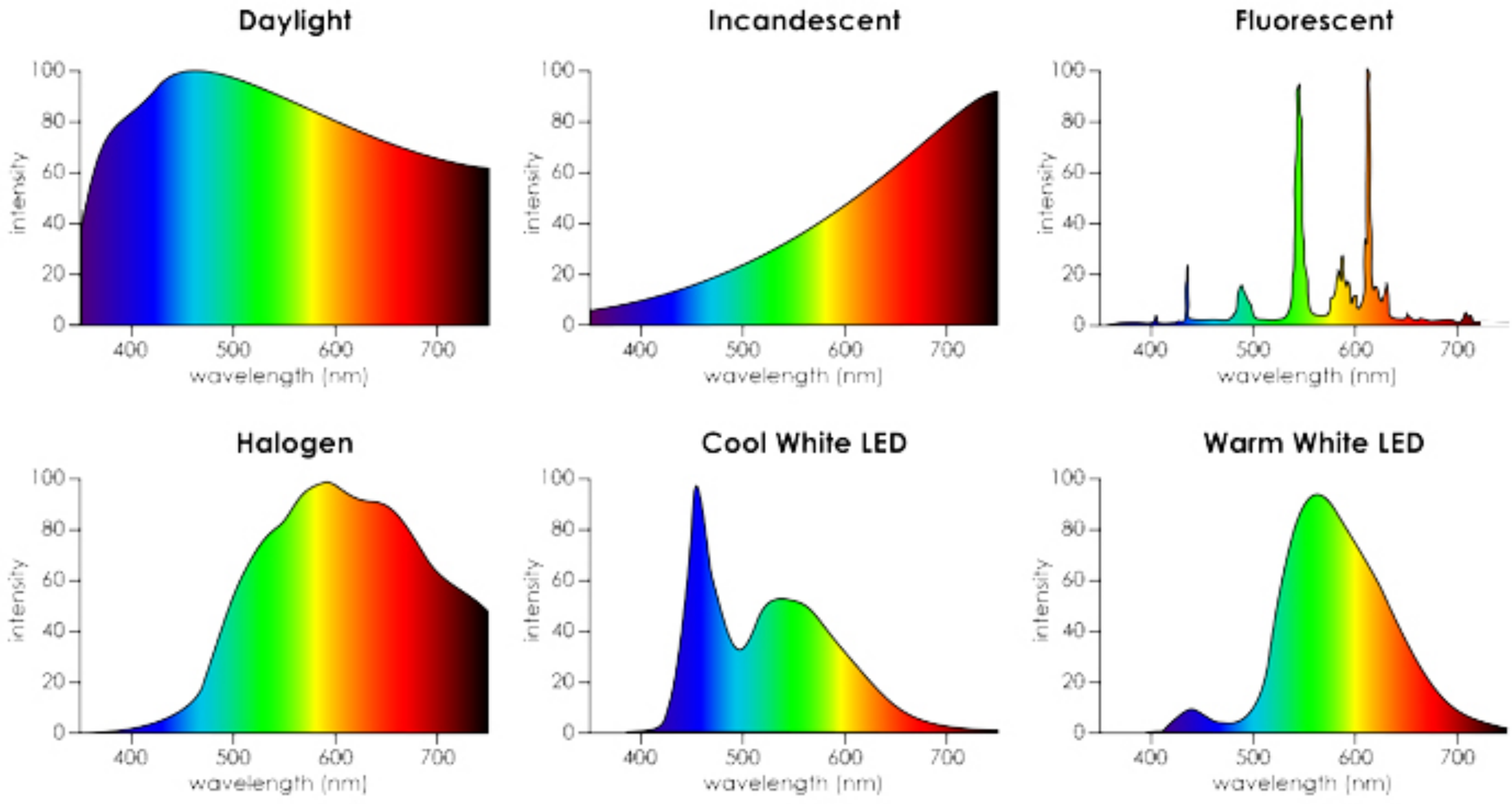


Figure credit:

admesy
ADVANCED MEASUREMENT SYSTEMS

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)

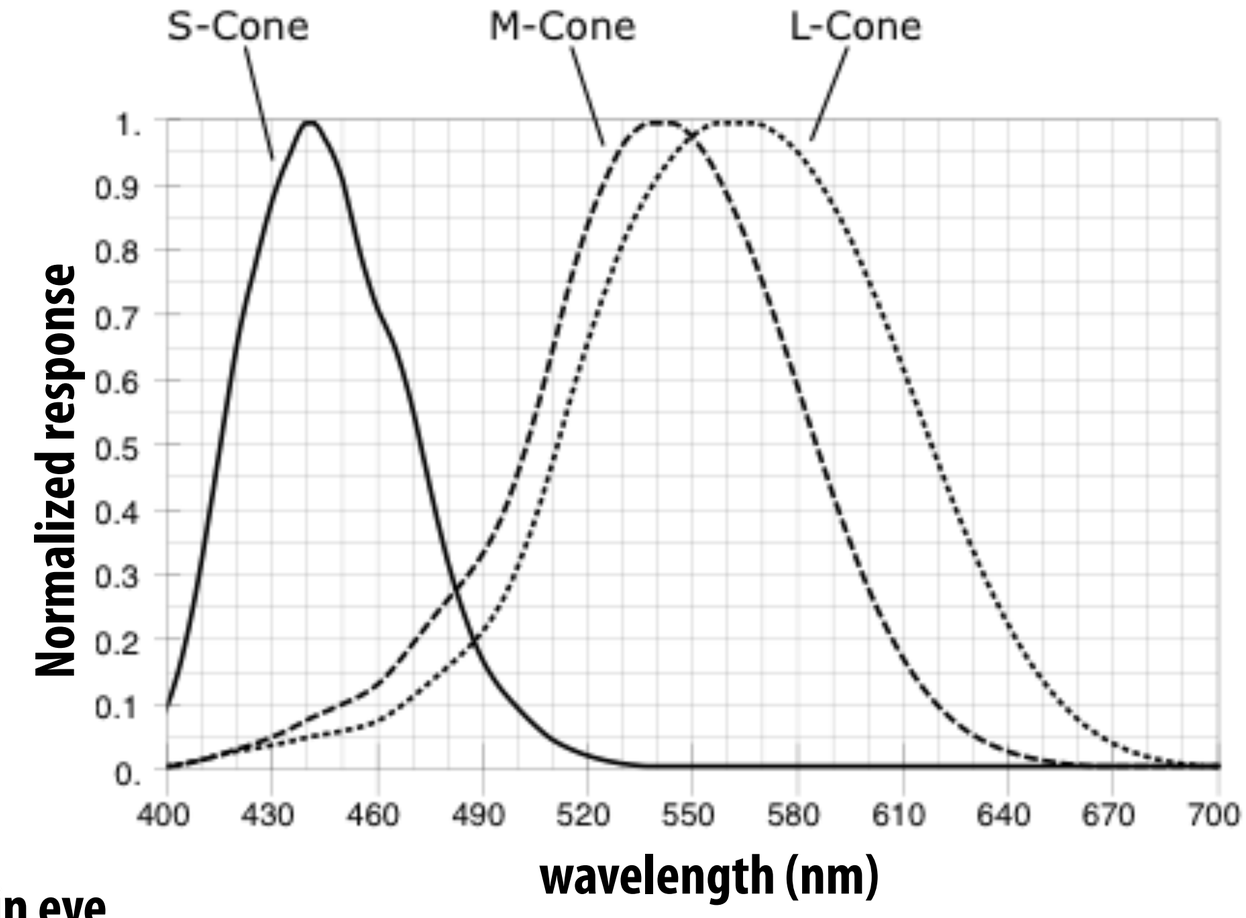
Takeaway: the space of human-perceivable colors is three dimensional

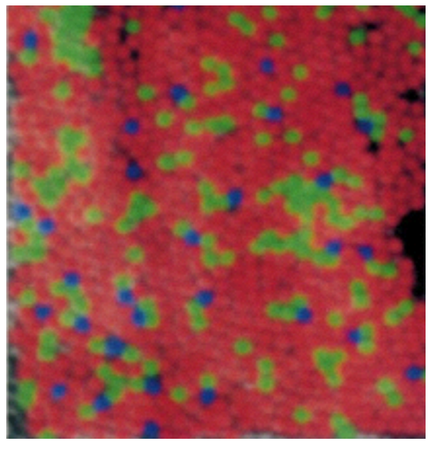
$$S = \int_\lambda \Phi(\lambda) S(\lambda) d\lambda$$

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

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

Response functions for S, M, and L cones

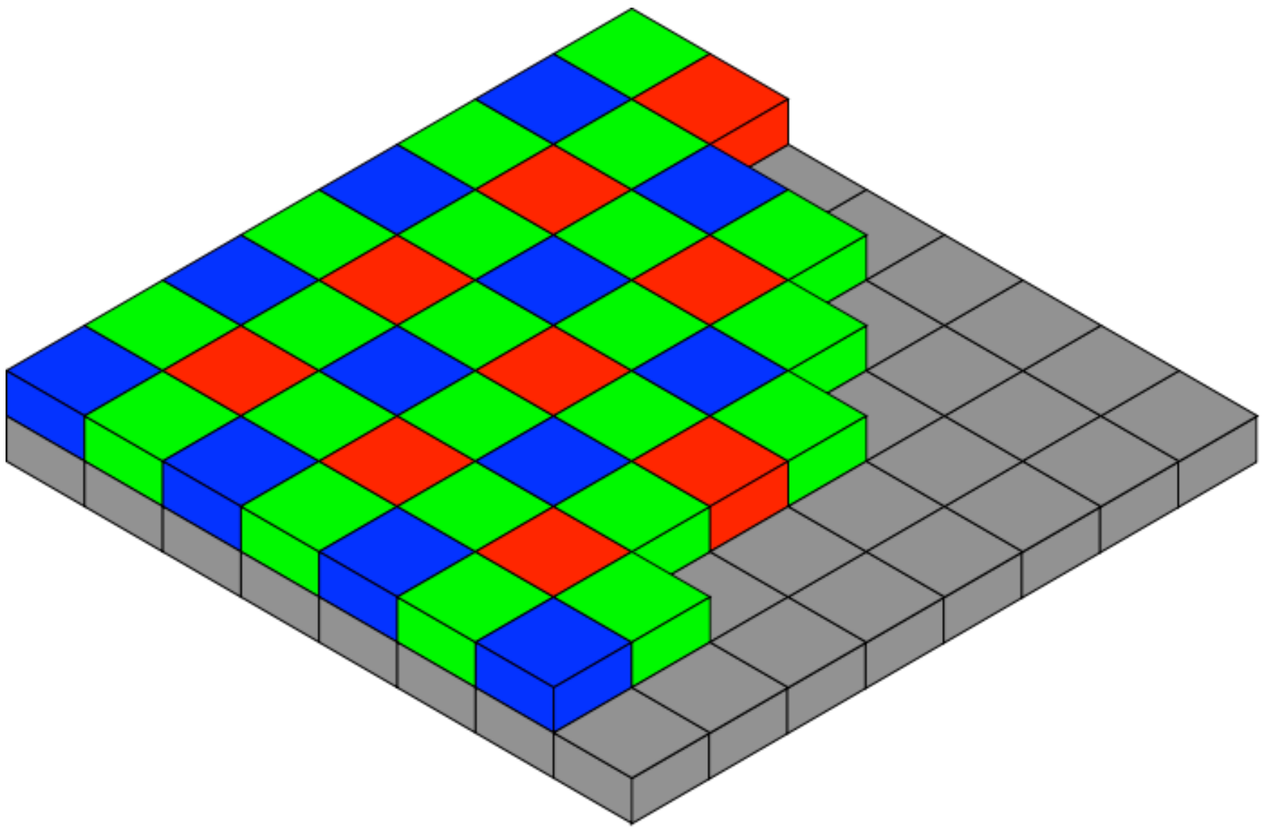




Uneven distribution of cone types in eye
~64% of cones are L cones, ~ 32% M cones

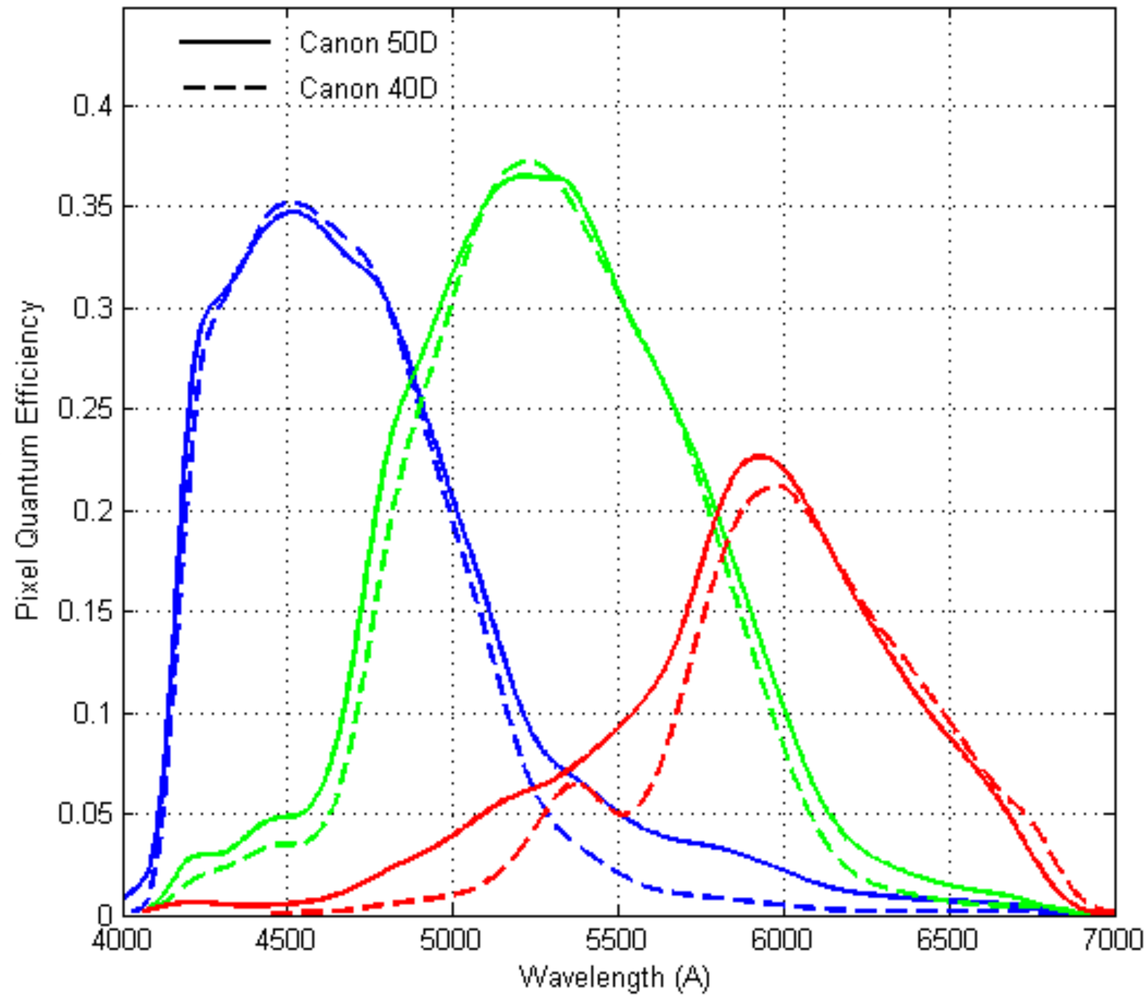
Color filter array (Bayer mosaic)

- 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

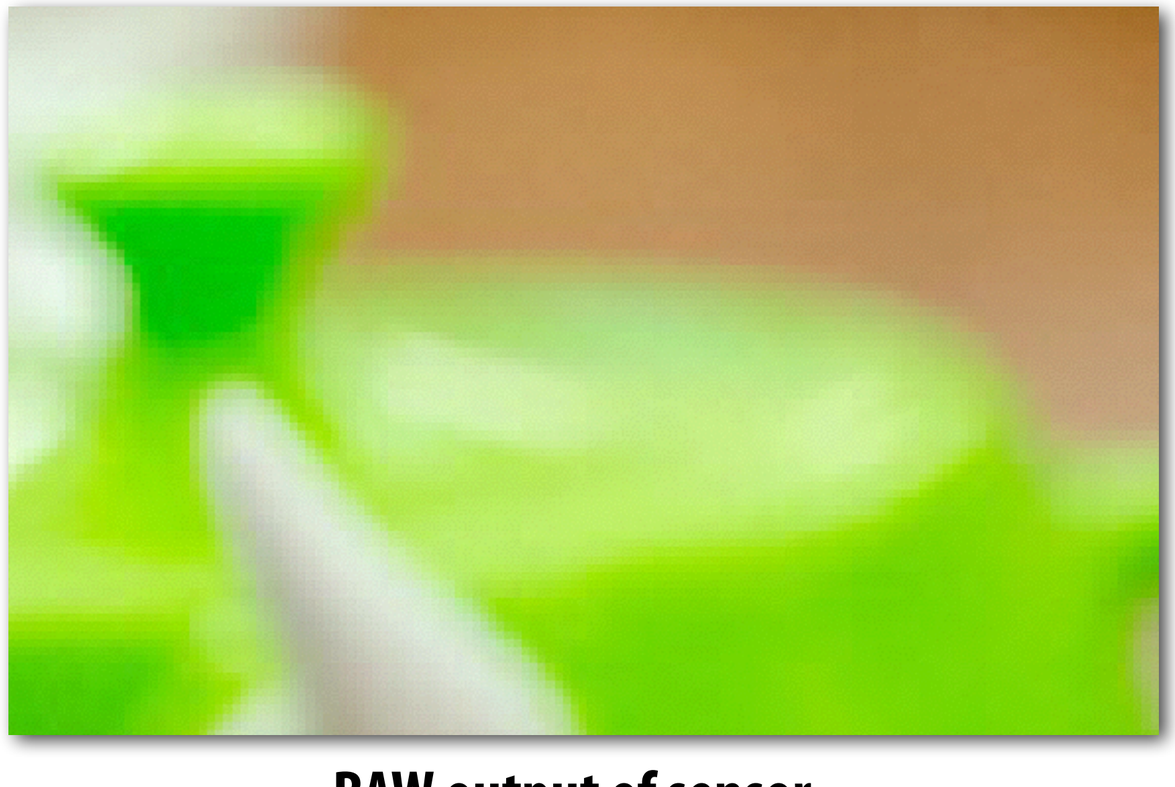


$f(\lambda)$

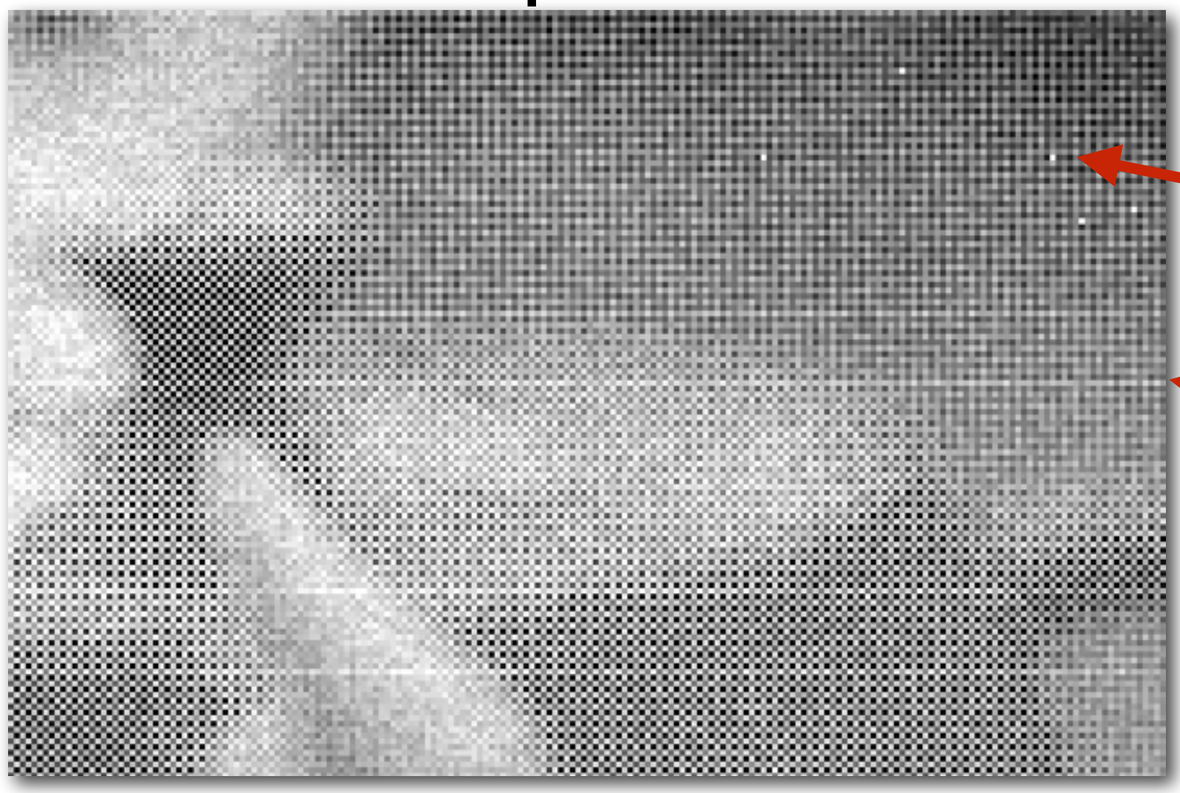
Image credit:
Wikipedia, Christian Buil (http://www.astrosurf.com/~buil/cameras.htm)

RAW sensor output (simulated data)

Light Hitting Sensor



RAW output of sensor



"Hot pixel"

Bad row

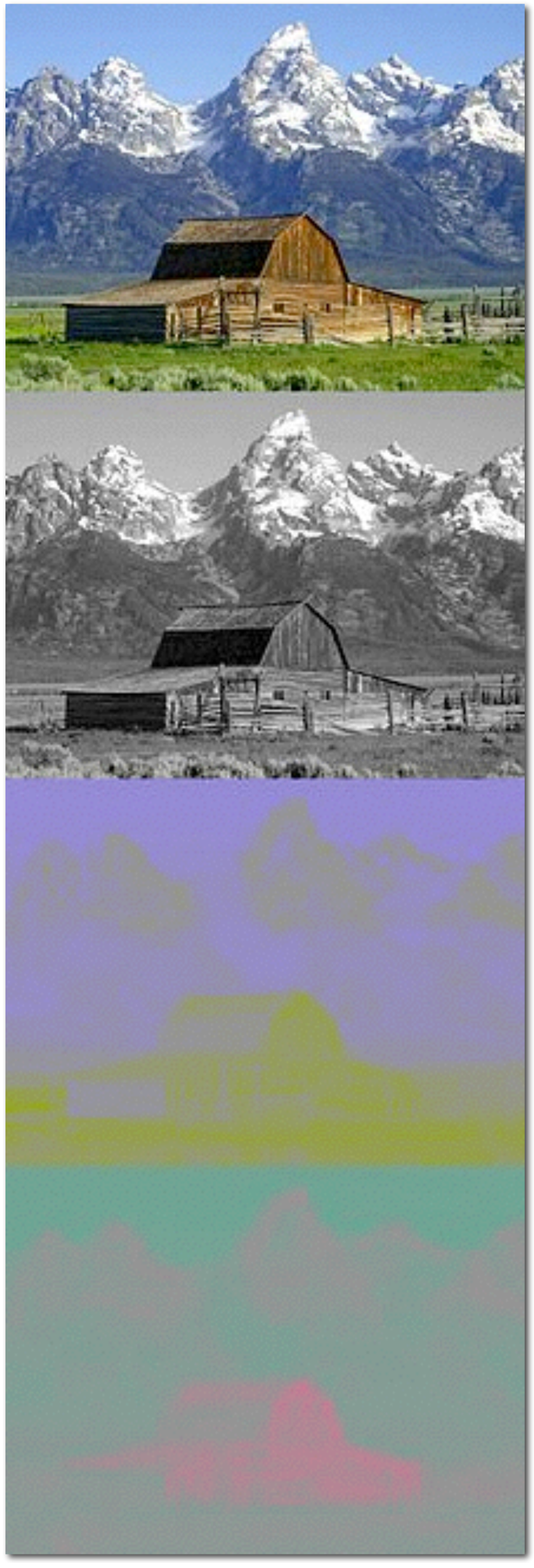
Another basis: Y′CbCr color space

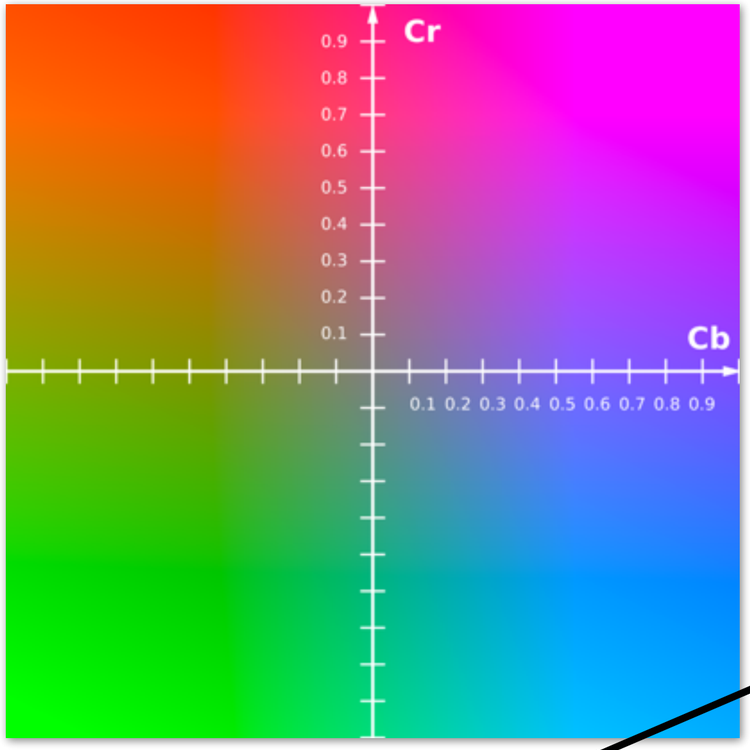
Recall: color is represented as a 3D value

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:

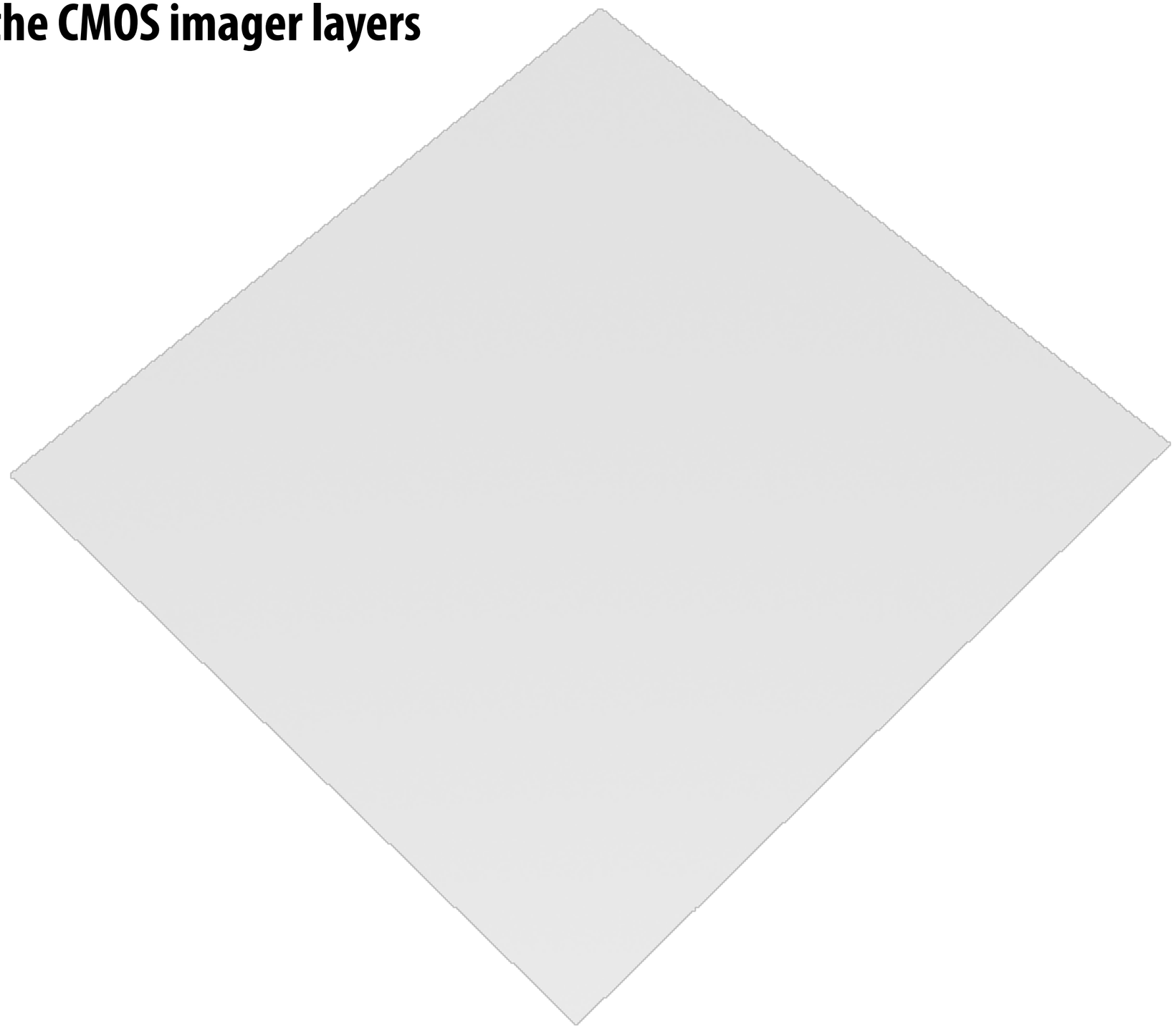
$$
\begin{aligned}
Y' &= 16 + \frac{65.738 \cdot R'_D}{256} + \frac{129.057 \cdot G'_D}{256} + \frac{25.064 \cdot B'_D}{256} \\
C_B &= 128 + \frac{-37.945 \cdot R'_D}{256} - \frac{74.494 \cdot G'_D}{256} + \frac{112.439 \cdot B'_D}{256} \\
C_R &= 128 + \frac{112.439 \cdot R'_D}{256} - \frac{94.154 \cdot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

Image credit: Wikipedia

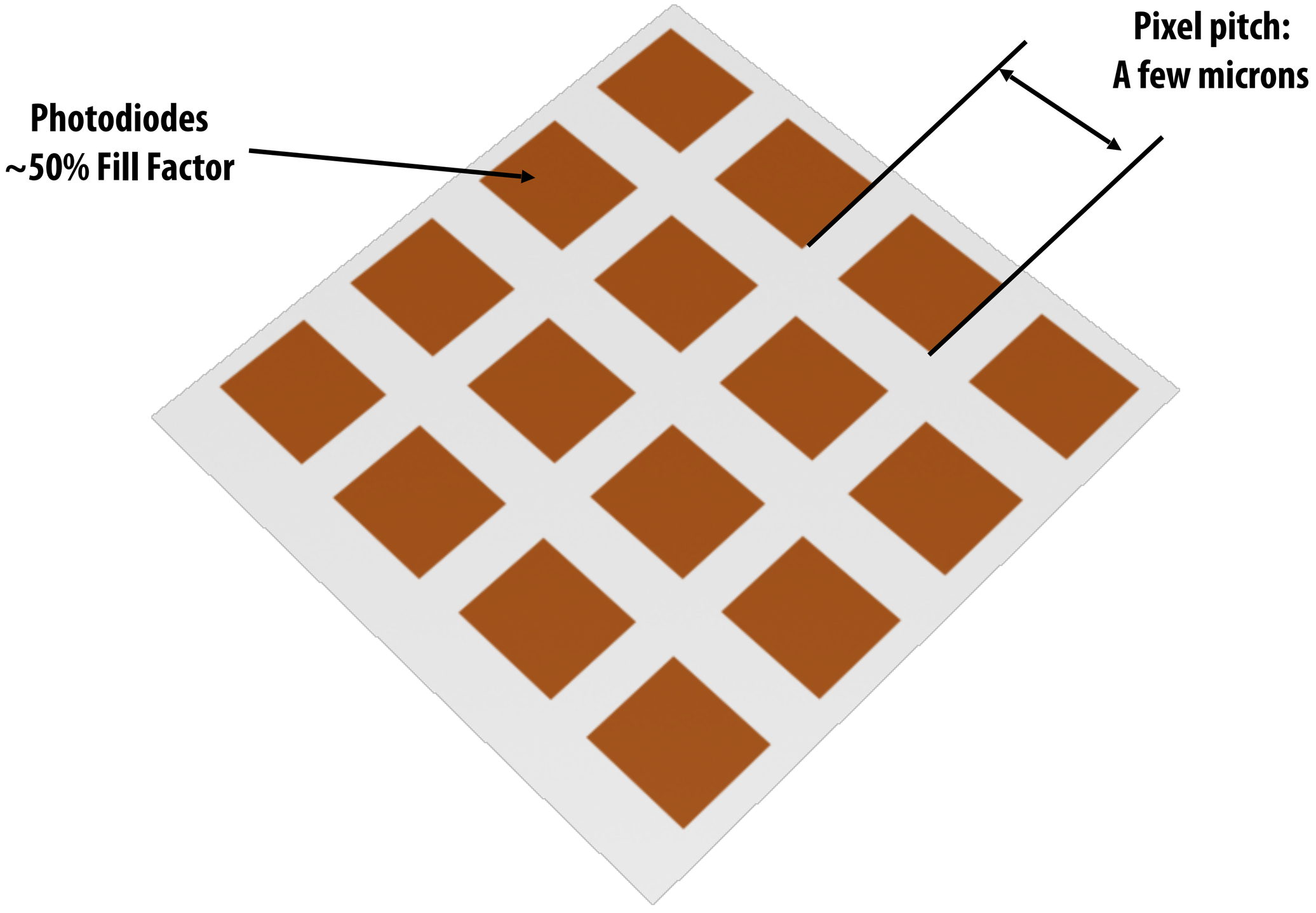
CMOS Pixel Structure

Front-side-illuminated (FSI) CMOS

Building up the CMOS imager layers

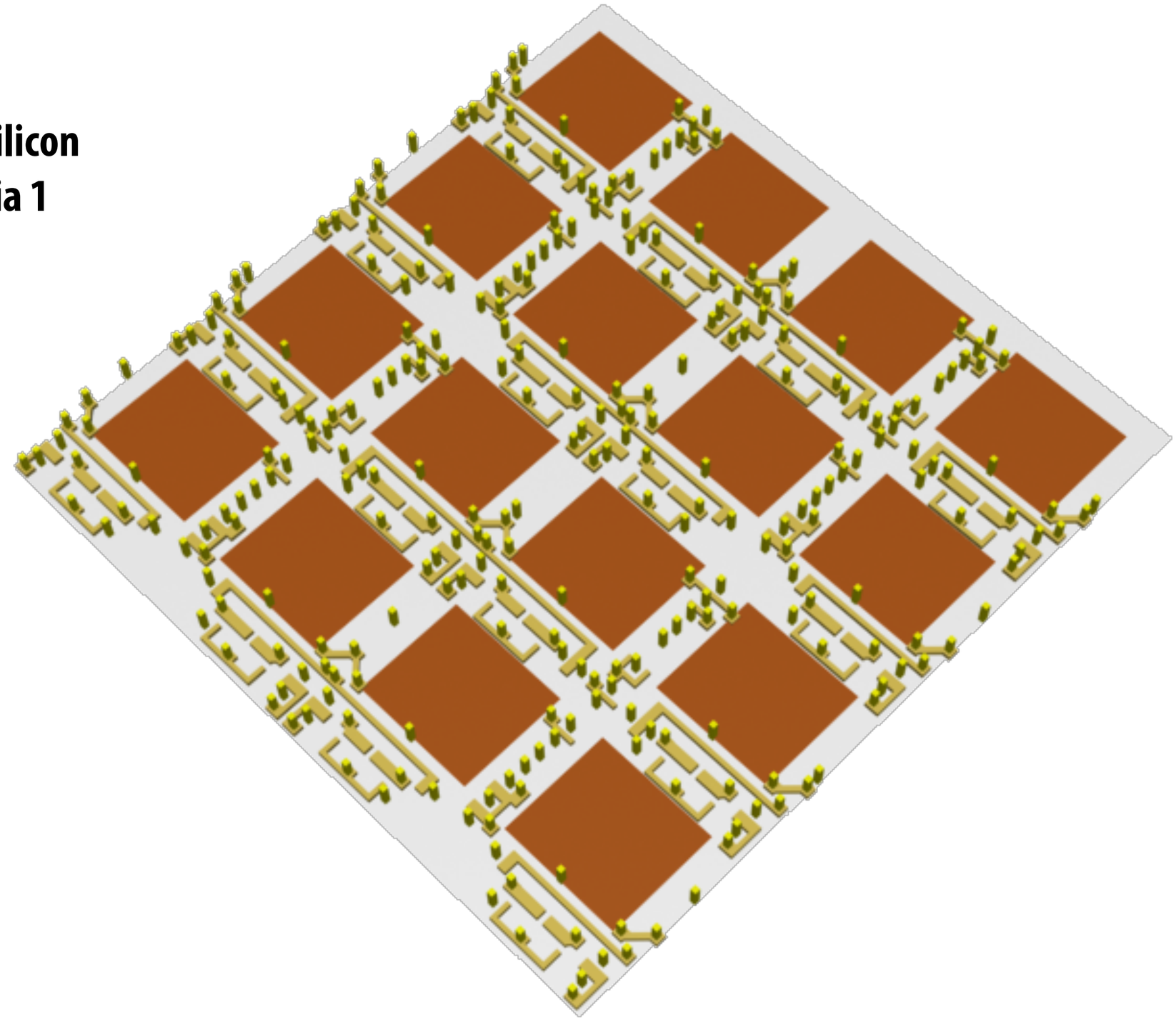


Courtesy R. Motta, Pixim



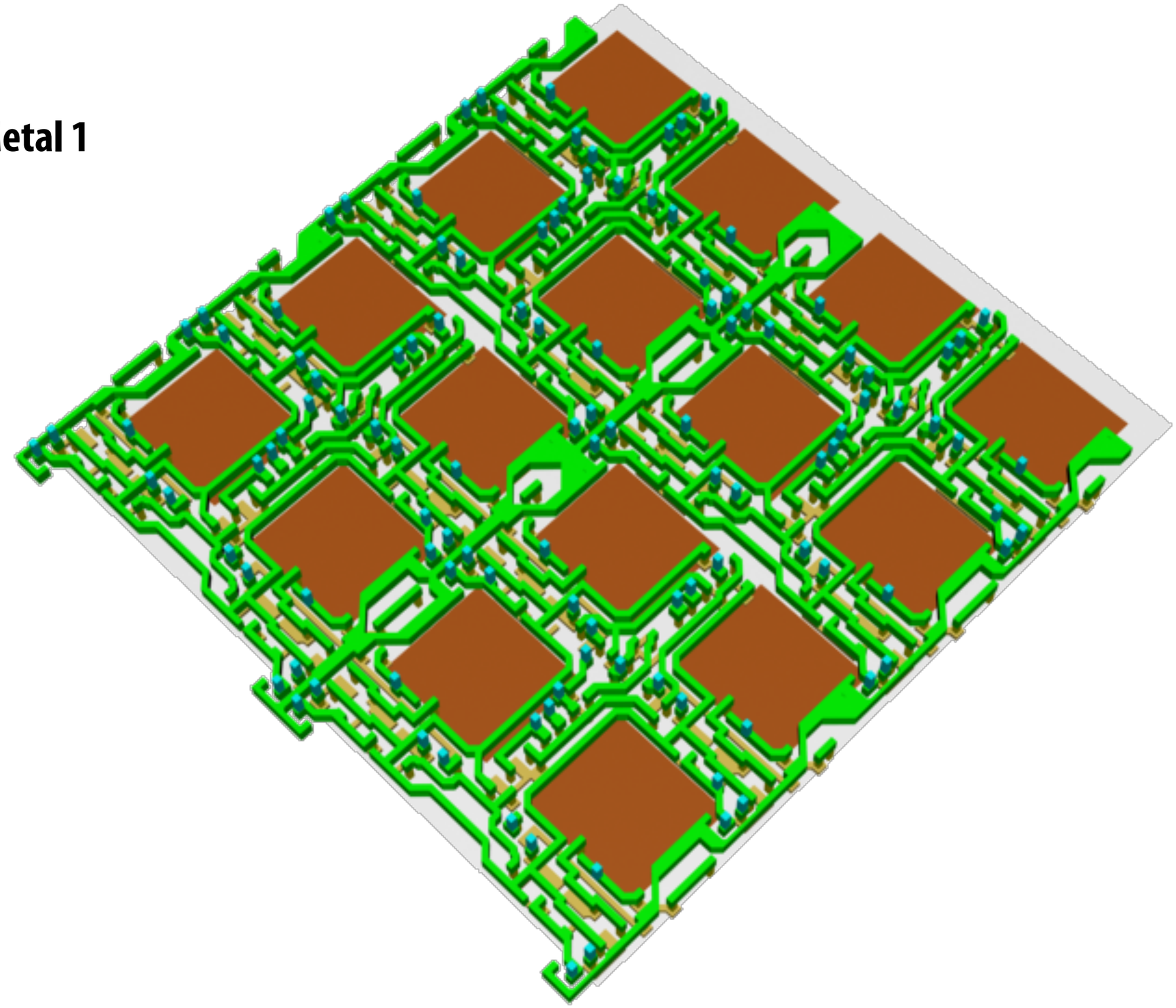
Courtesy R. Motta, Pixim

Polysilicon & Via 1



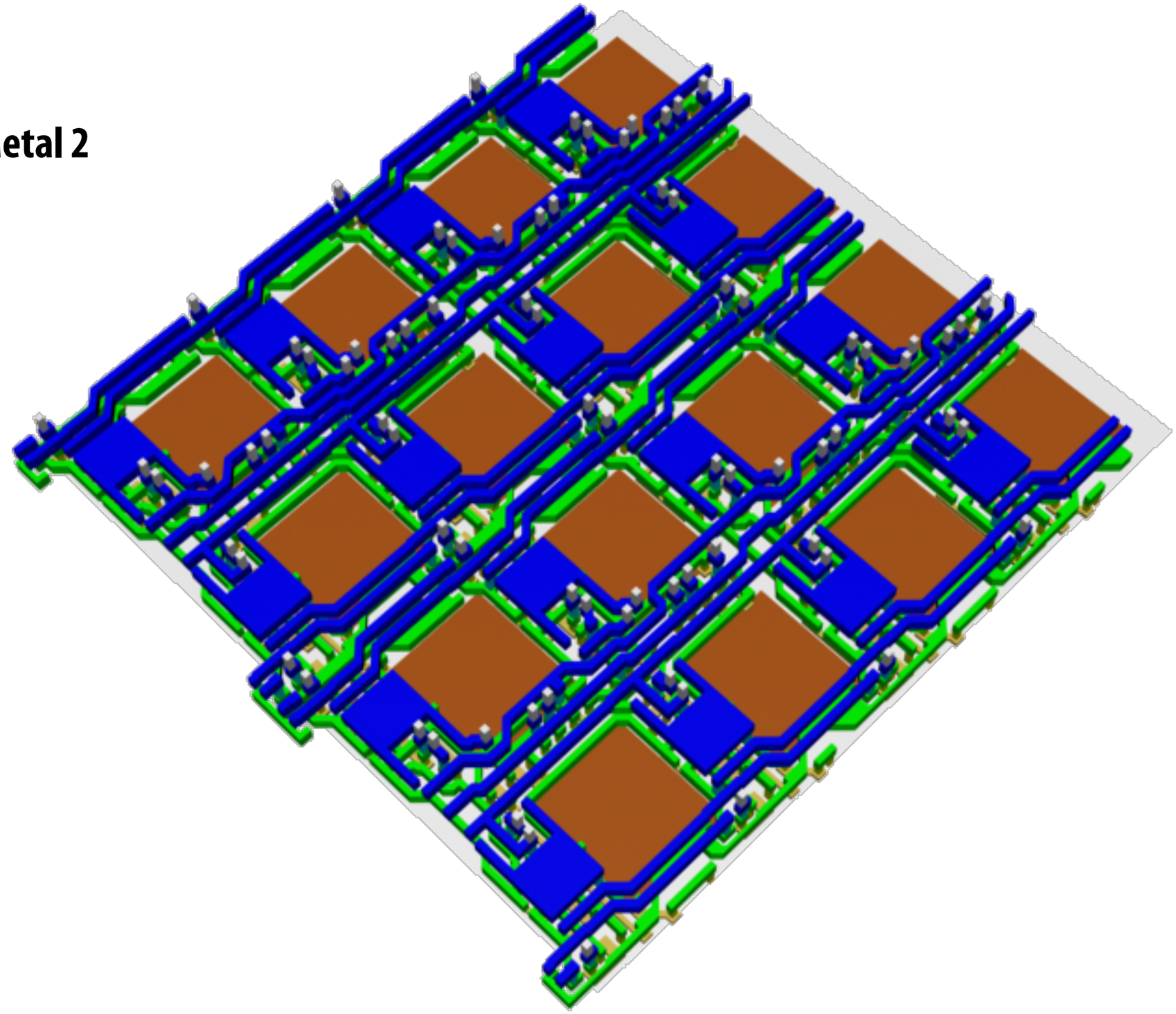
Courtesy R. Motta, Pixim

Metal 1



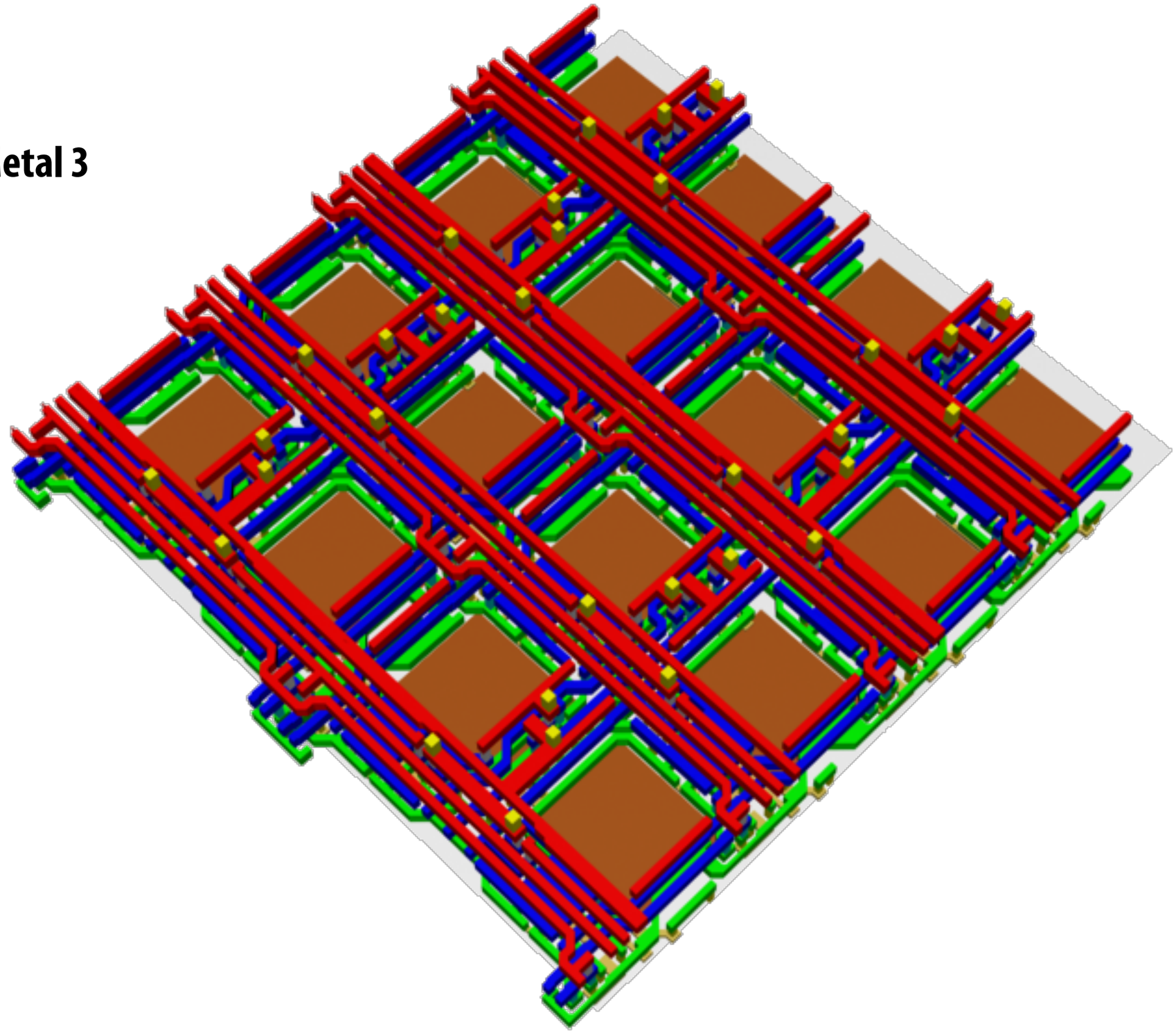
Courtesy R. Motta, Pixim

Metal 2



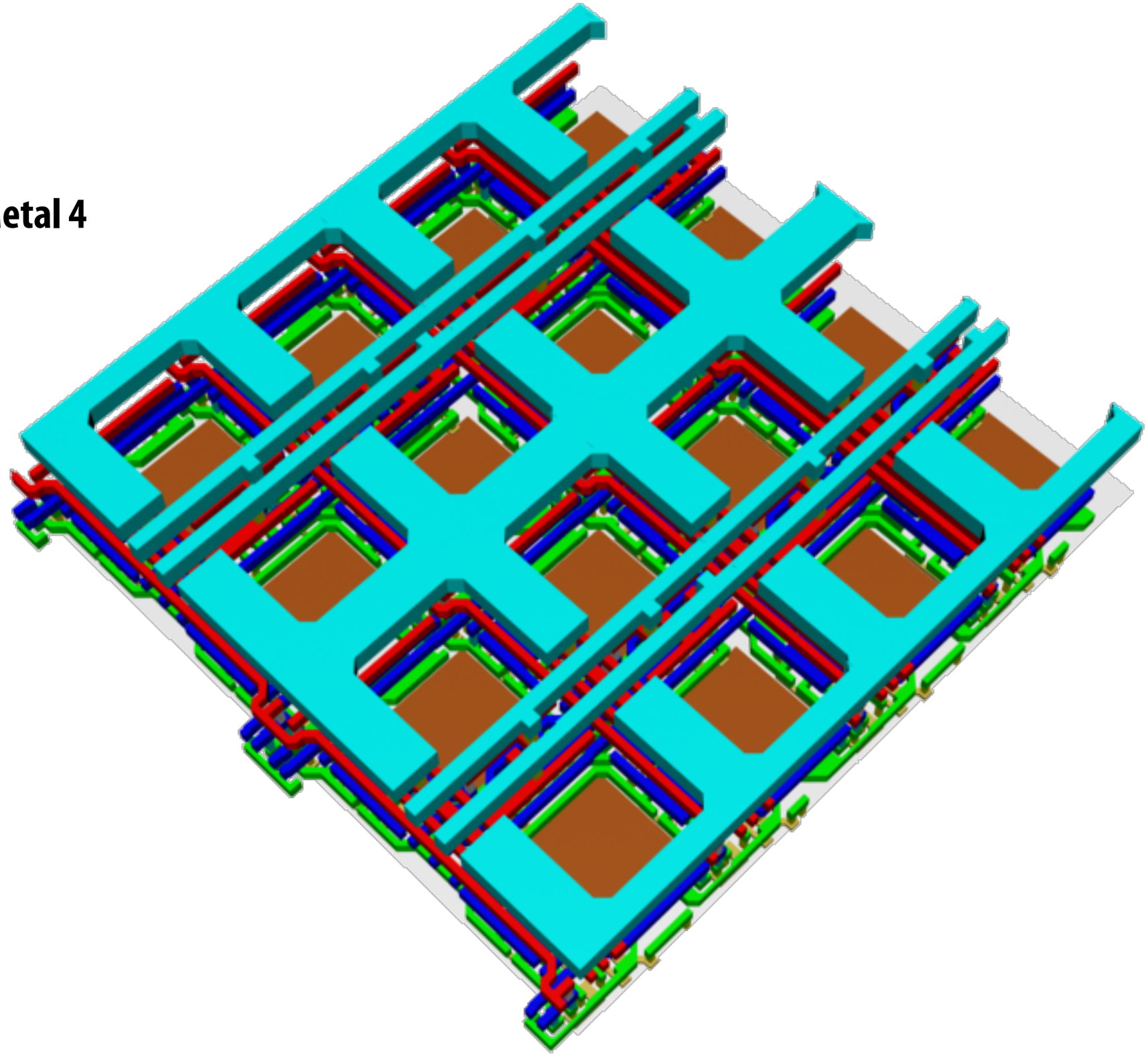
Courtesy R. Motta, Pixim

Metal 3



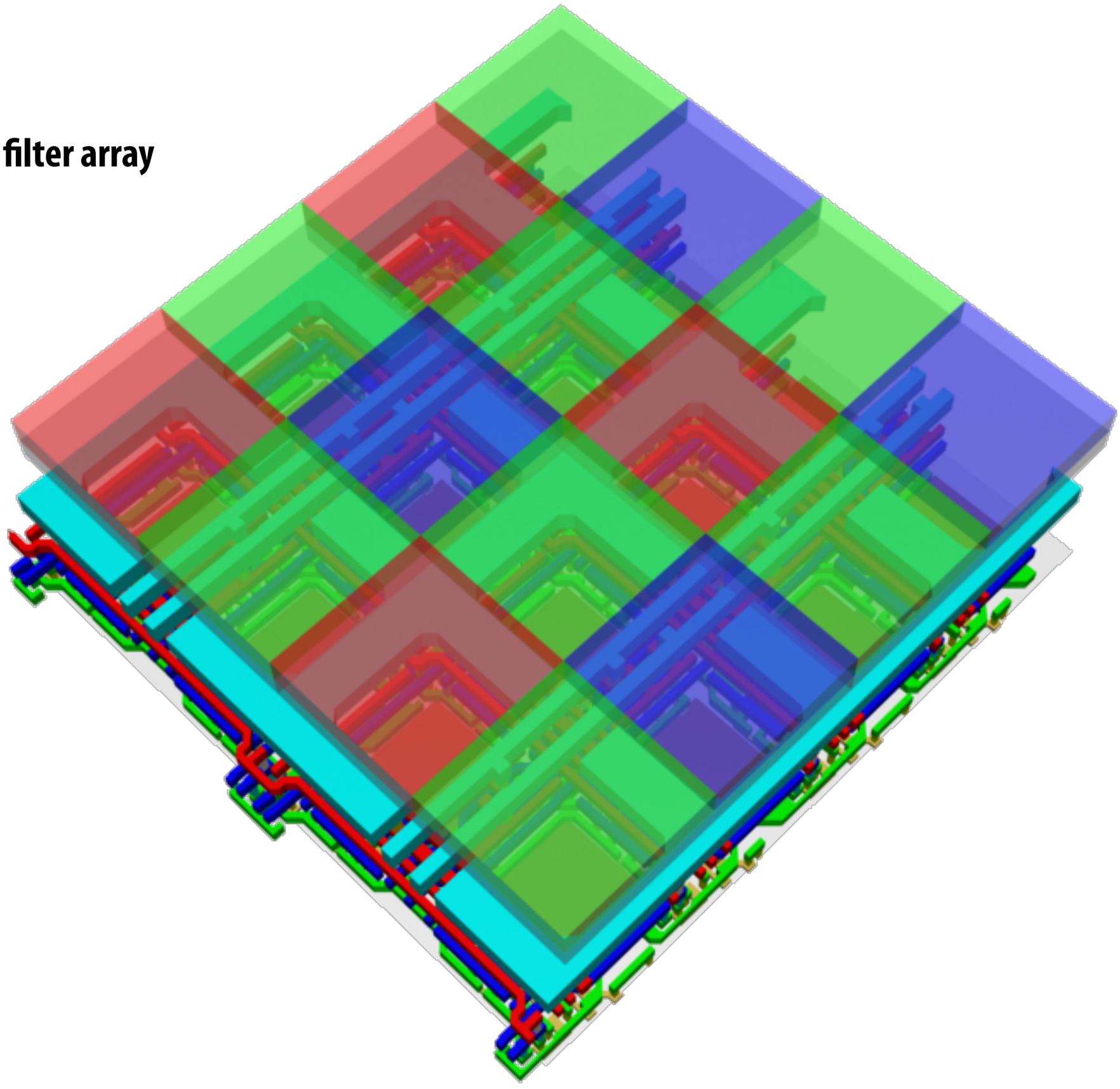
Courtesy R. Motta, Pixim

Metal 4



Courtesy R. Motta, Pixim

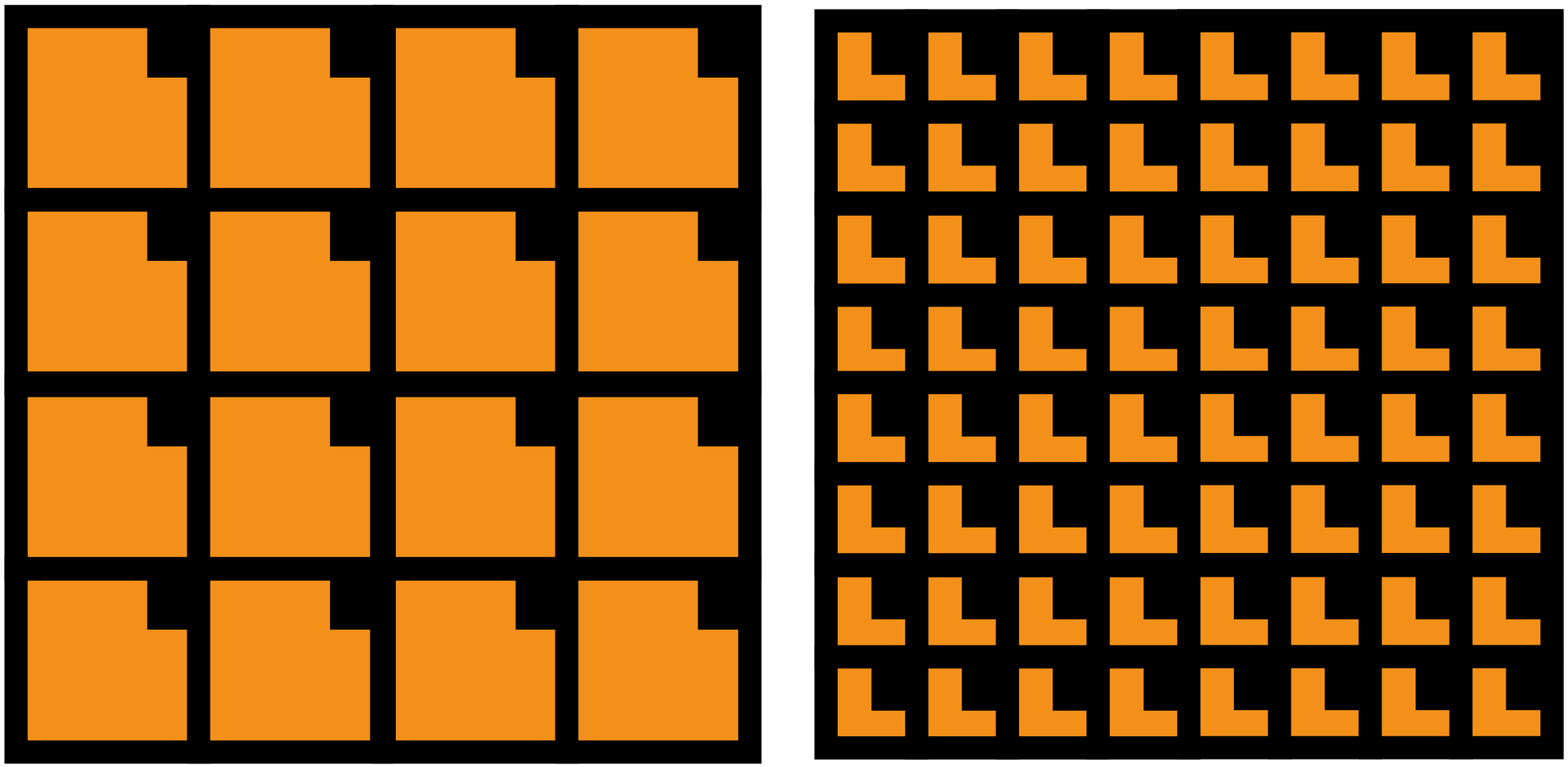
Color filter array



Courtesy R. Motta, Pixim

Pixel fill factor

Fraction of pixel area that integrates incoming light



Slide credit: Ren Ng

CMOS sensor pixel

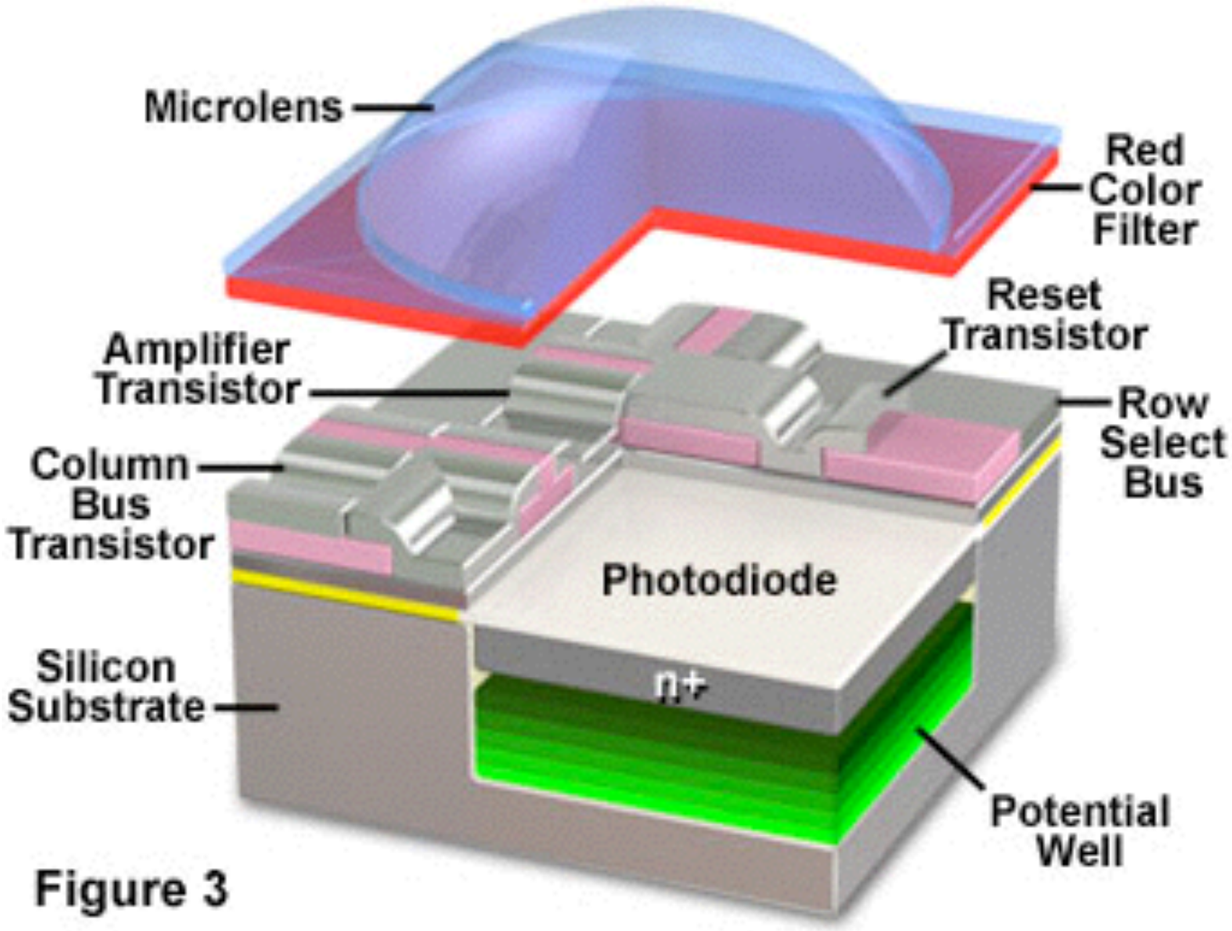


Figure 3

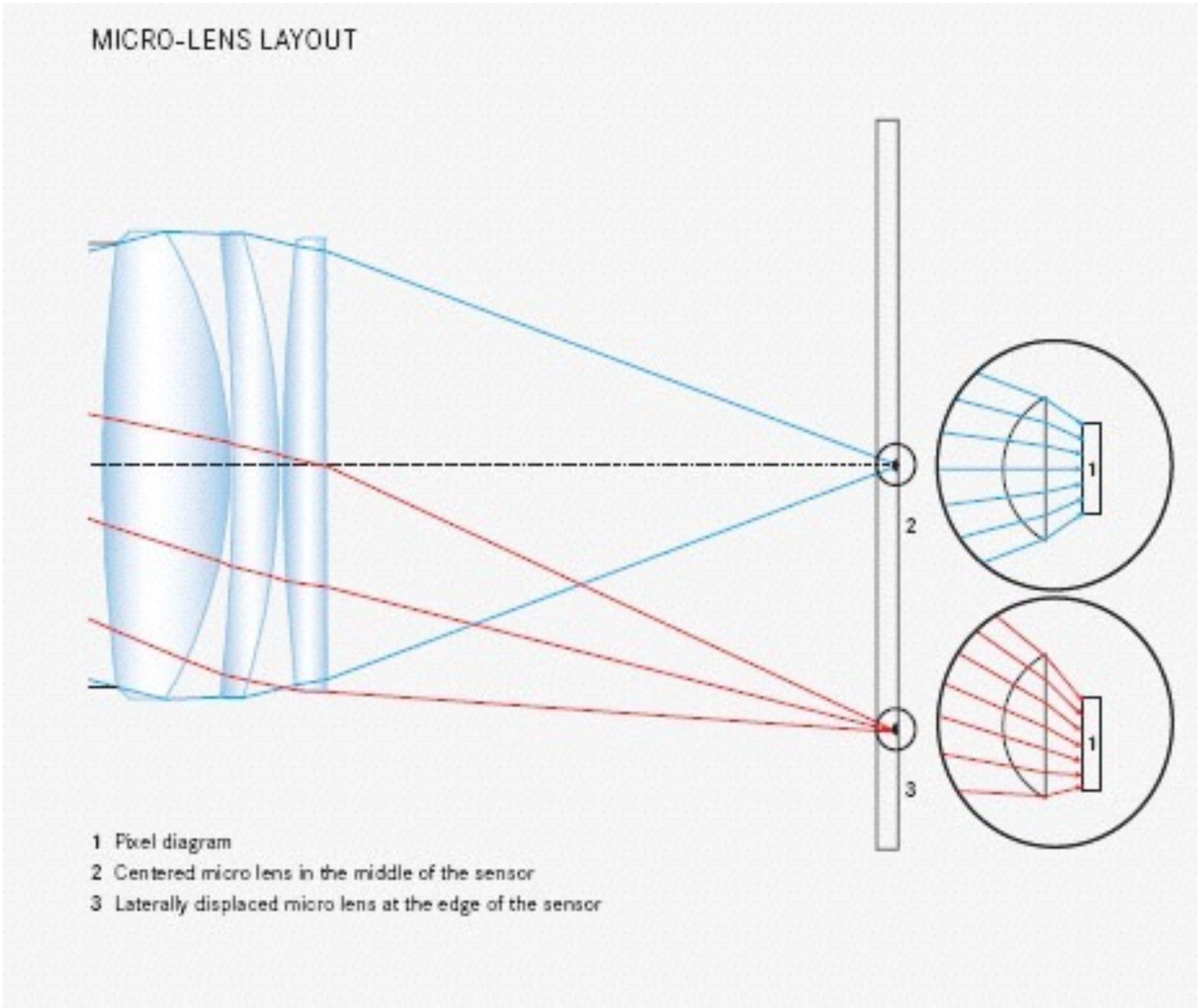
Color filter attenuates light

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

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

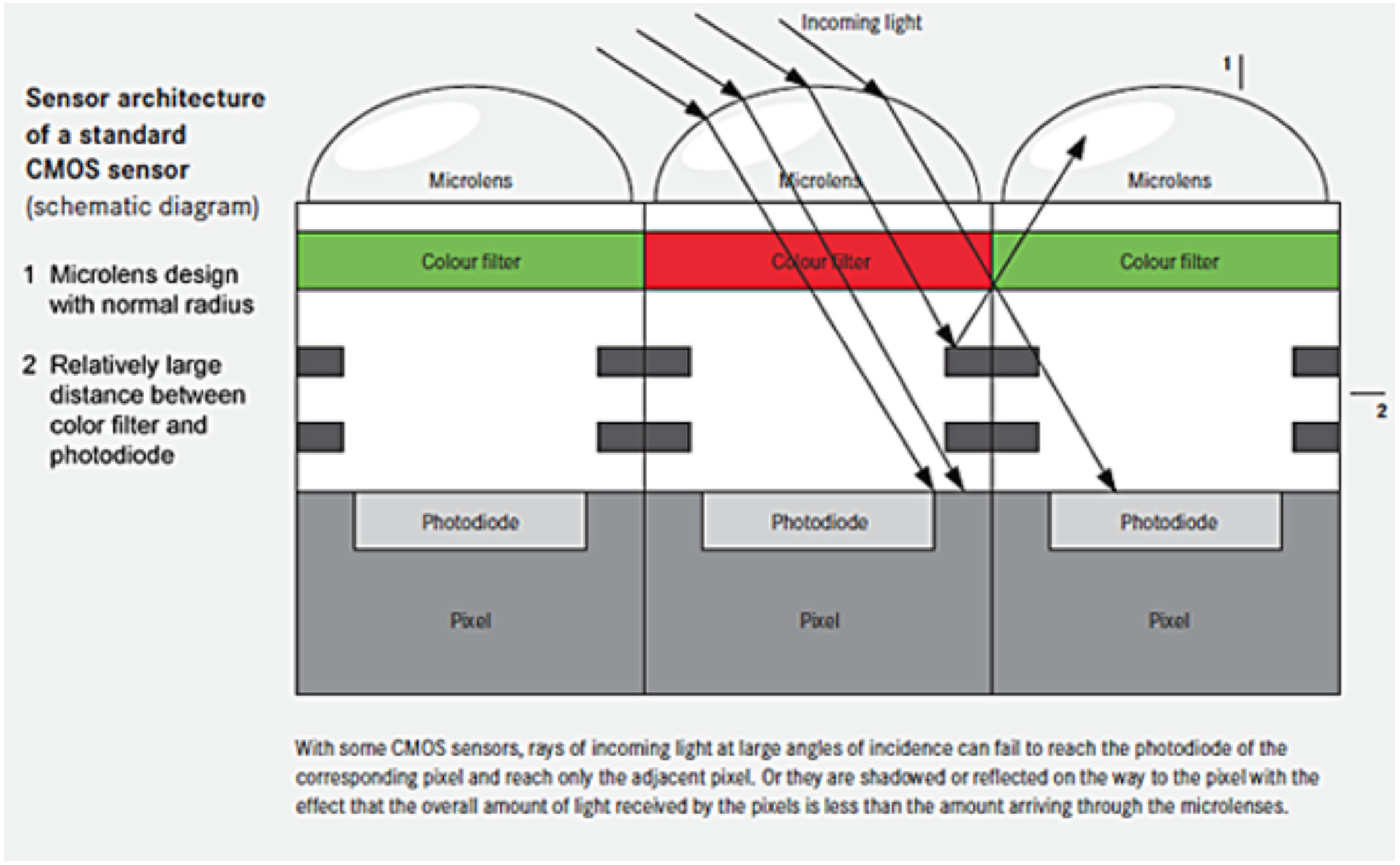


Shifted microlenses on M9 sensor.

Leica M9

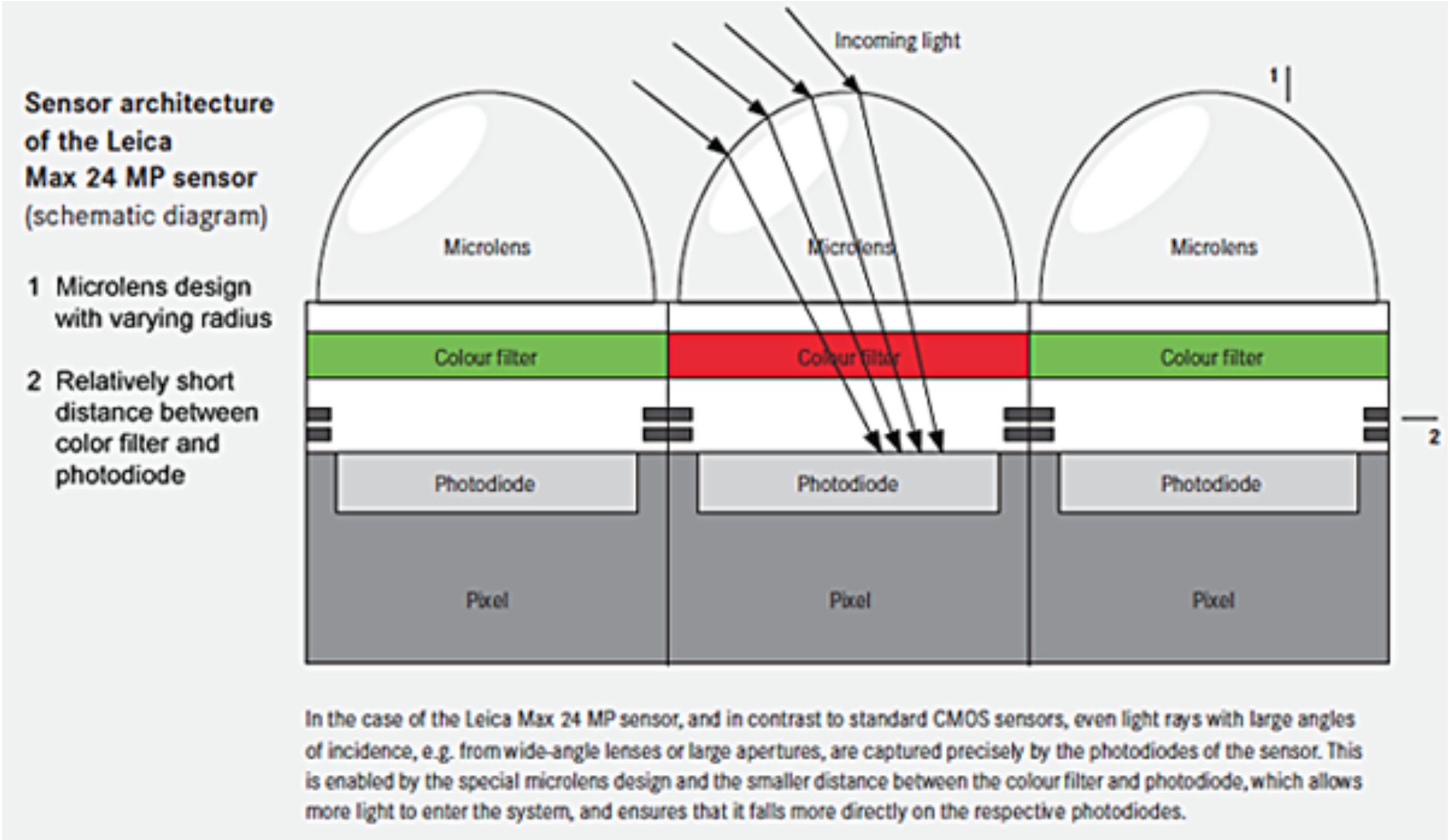
Slide credit: Ren Ng

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)

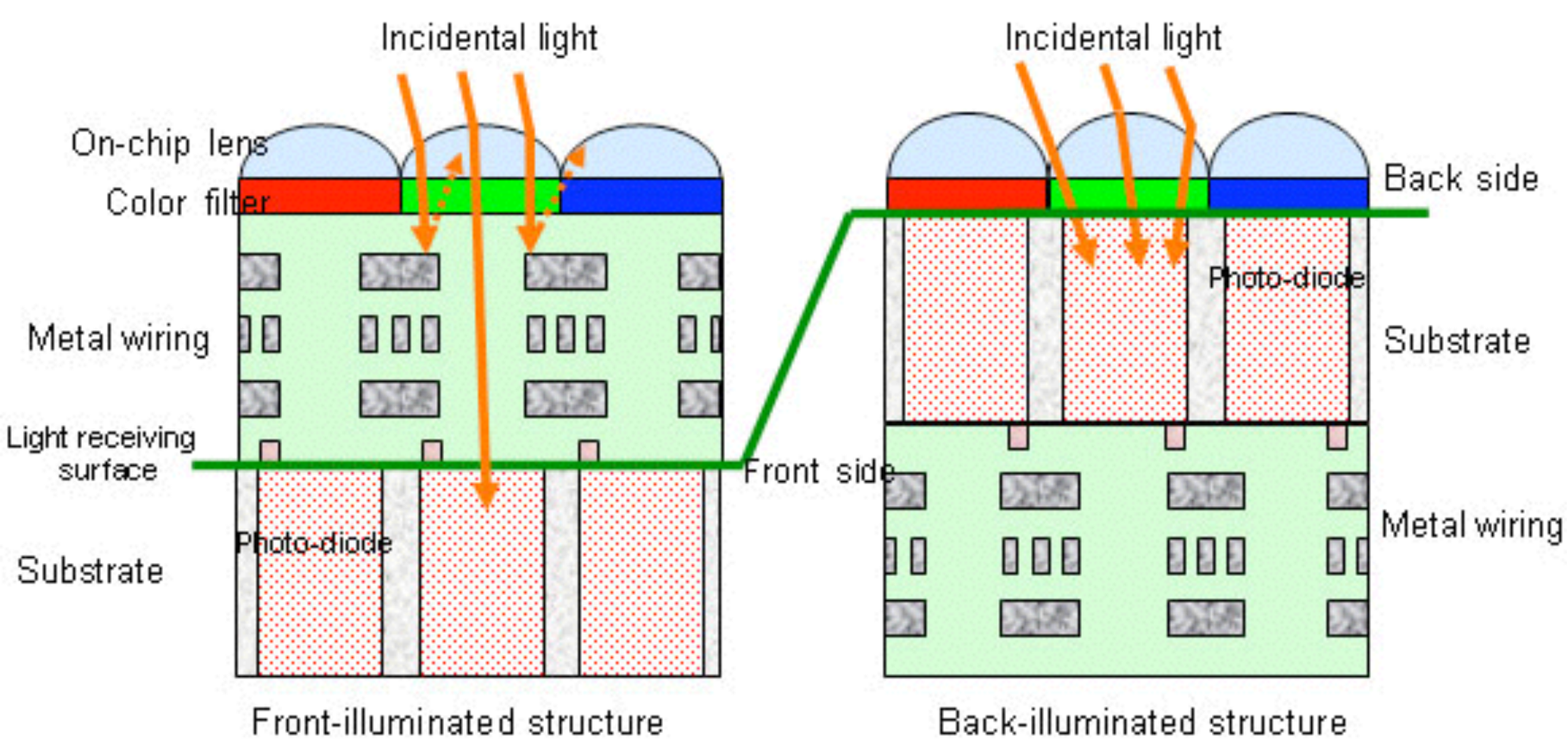
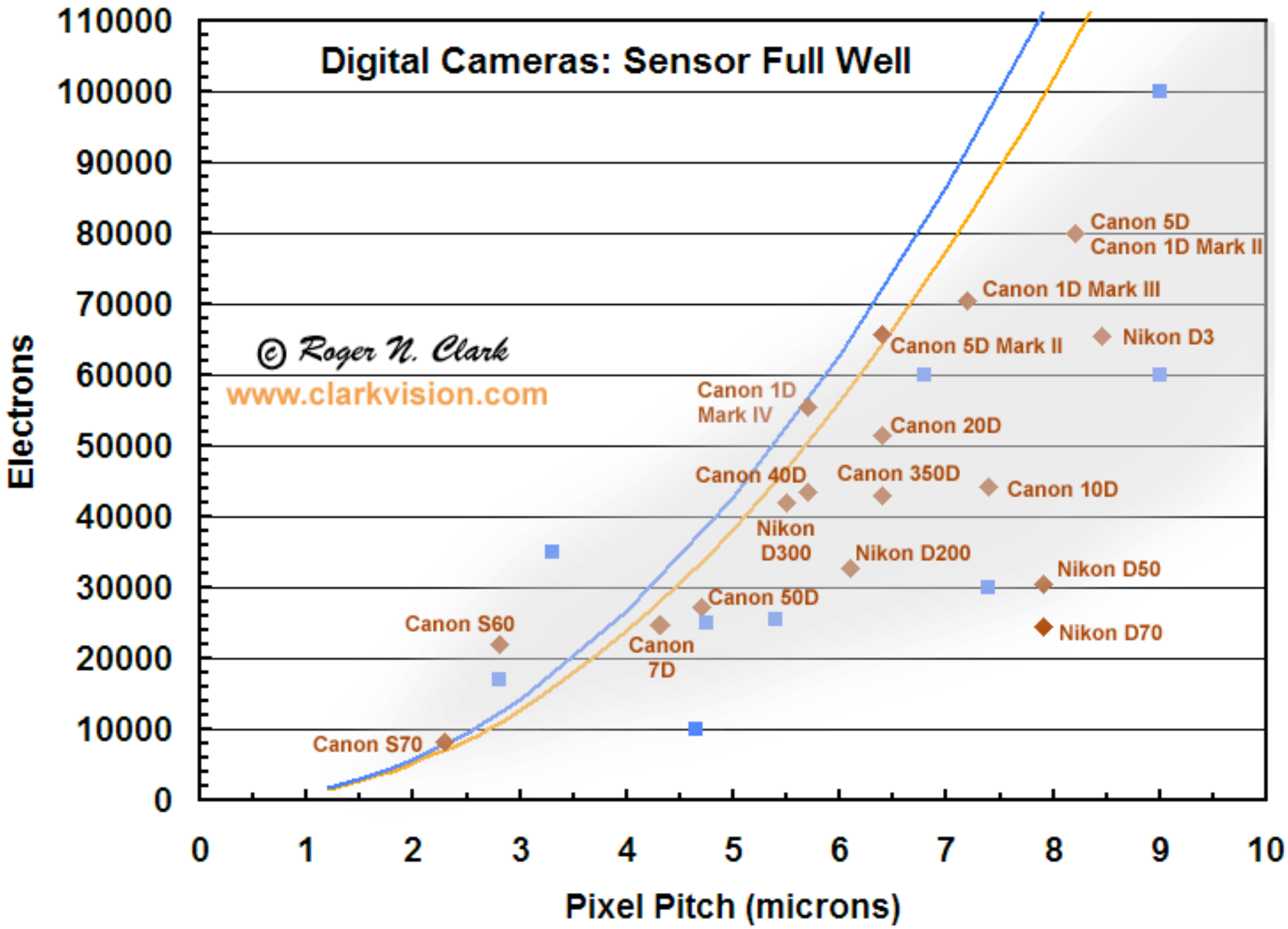


Illustration credit: Sony

Full-well capacity

Pixel saturates when photon capacity is exceeded

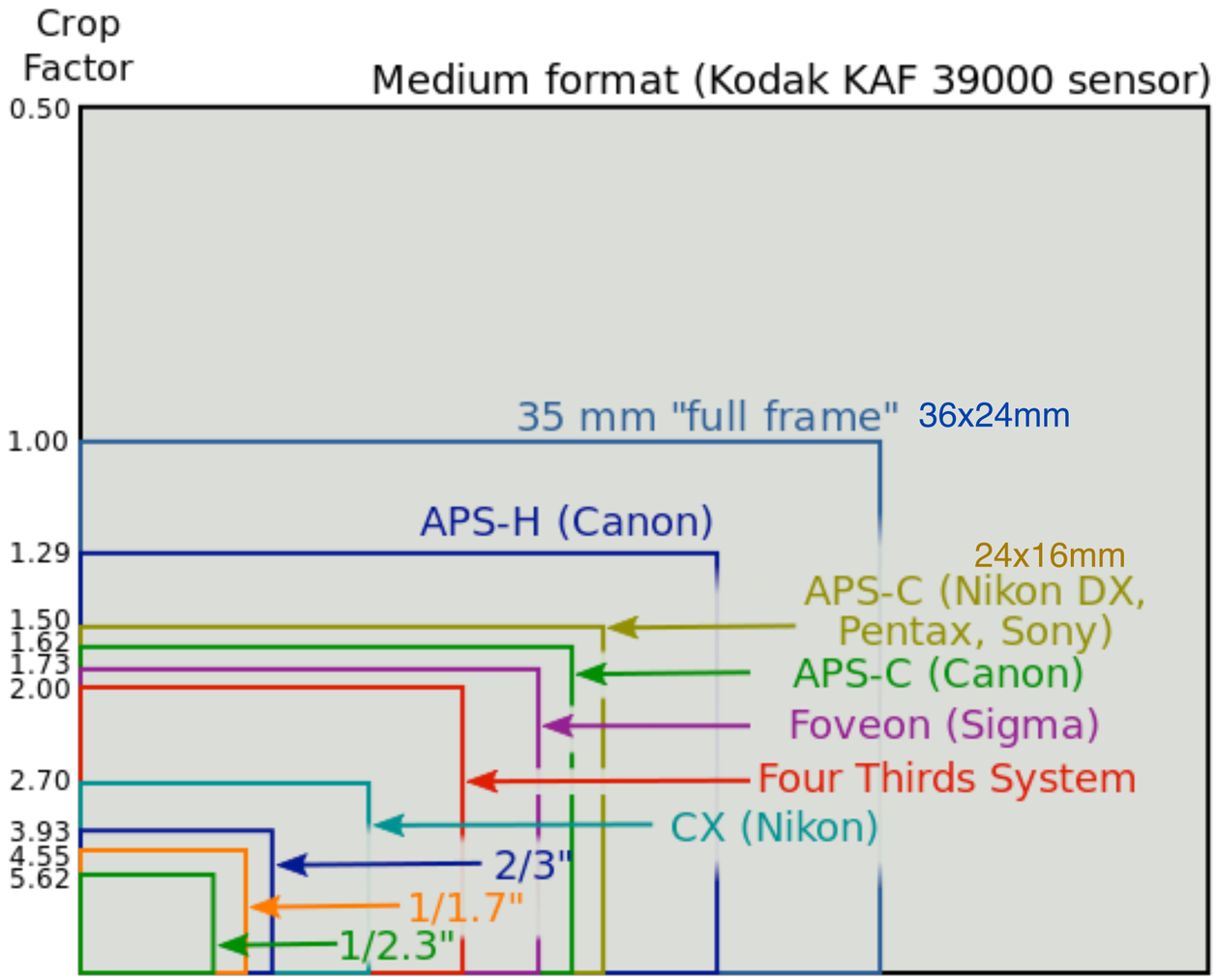


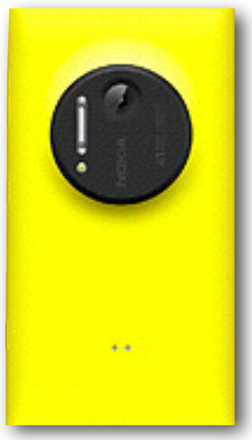
Graph credit: clarkvision.com

Oversaturated pixels

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

Photo of red square, moving to right

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

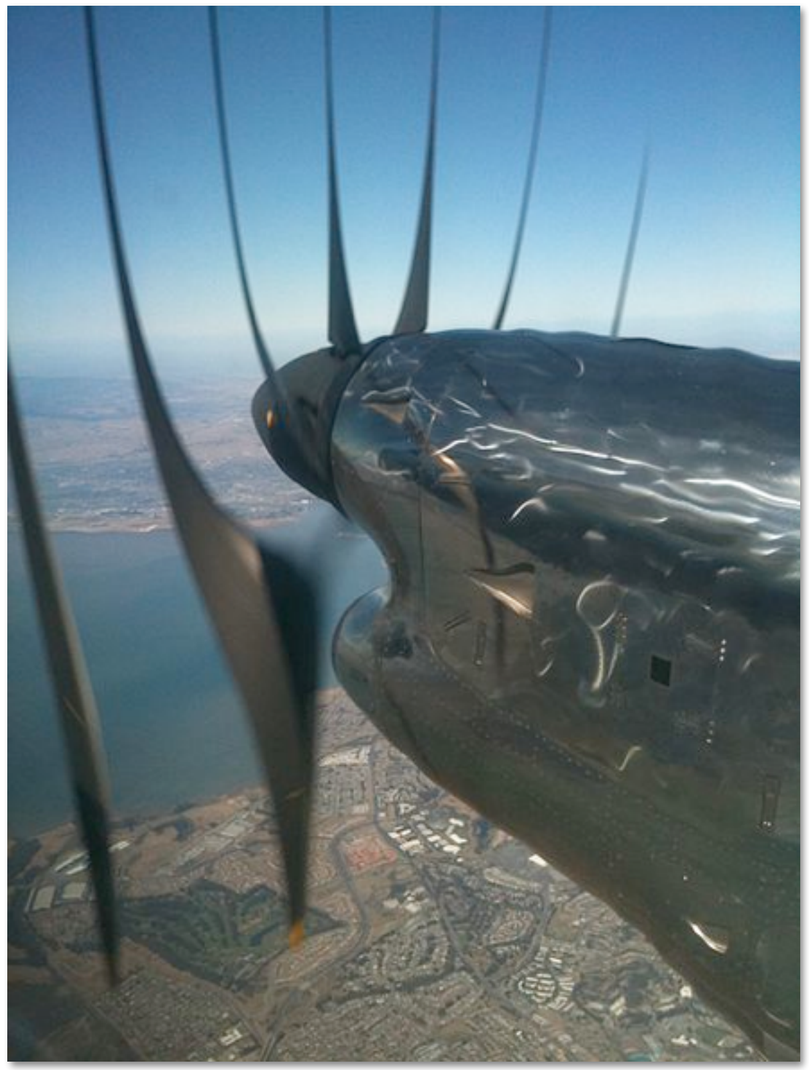


Image credit: Wikipedia

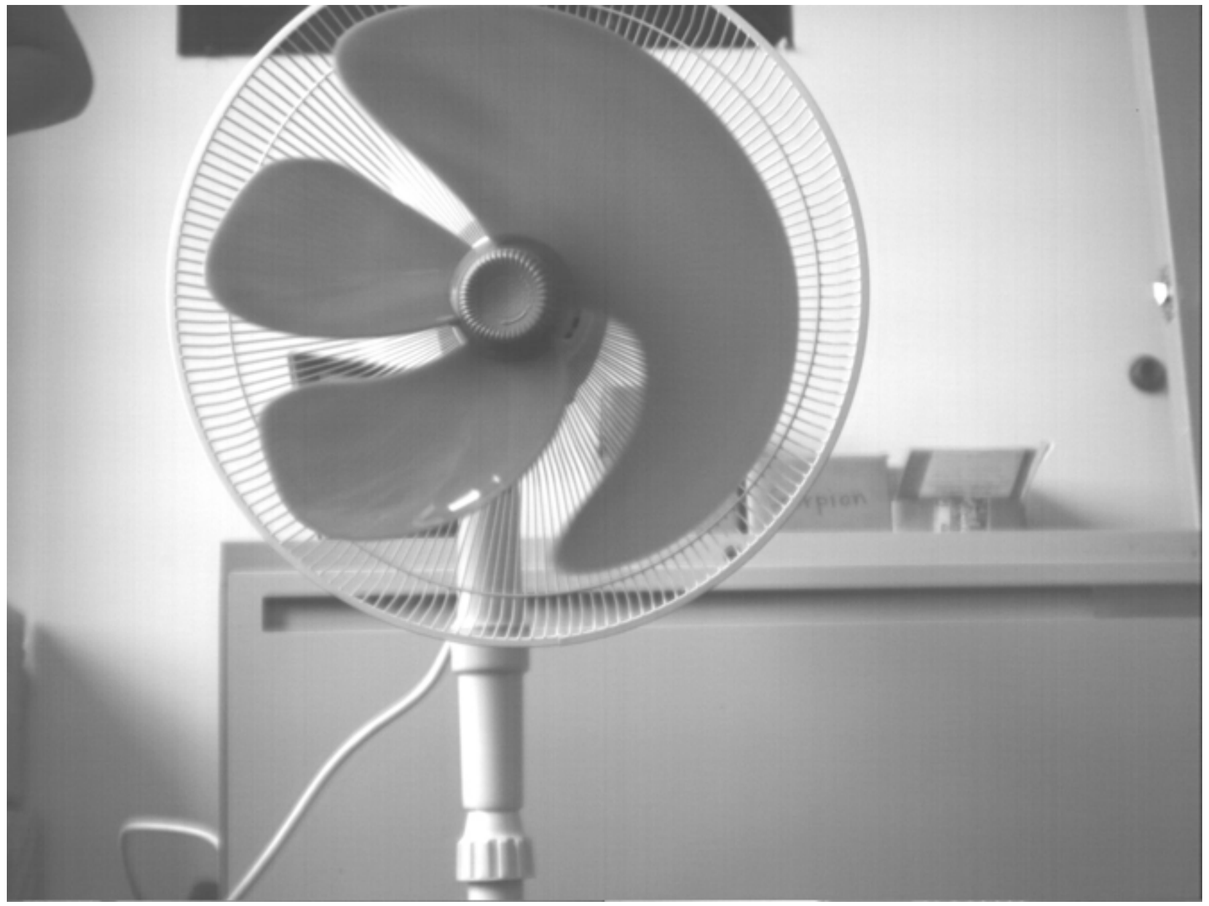
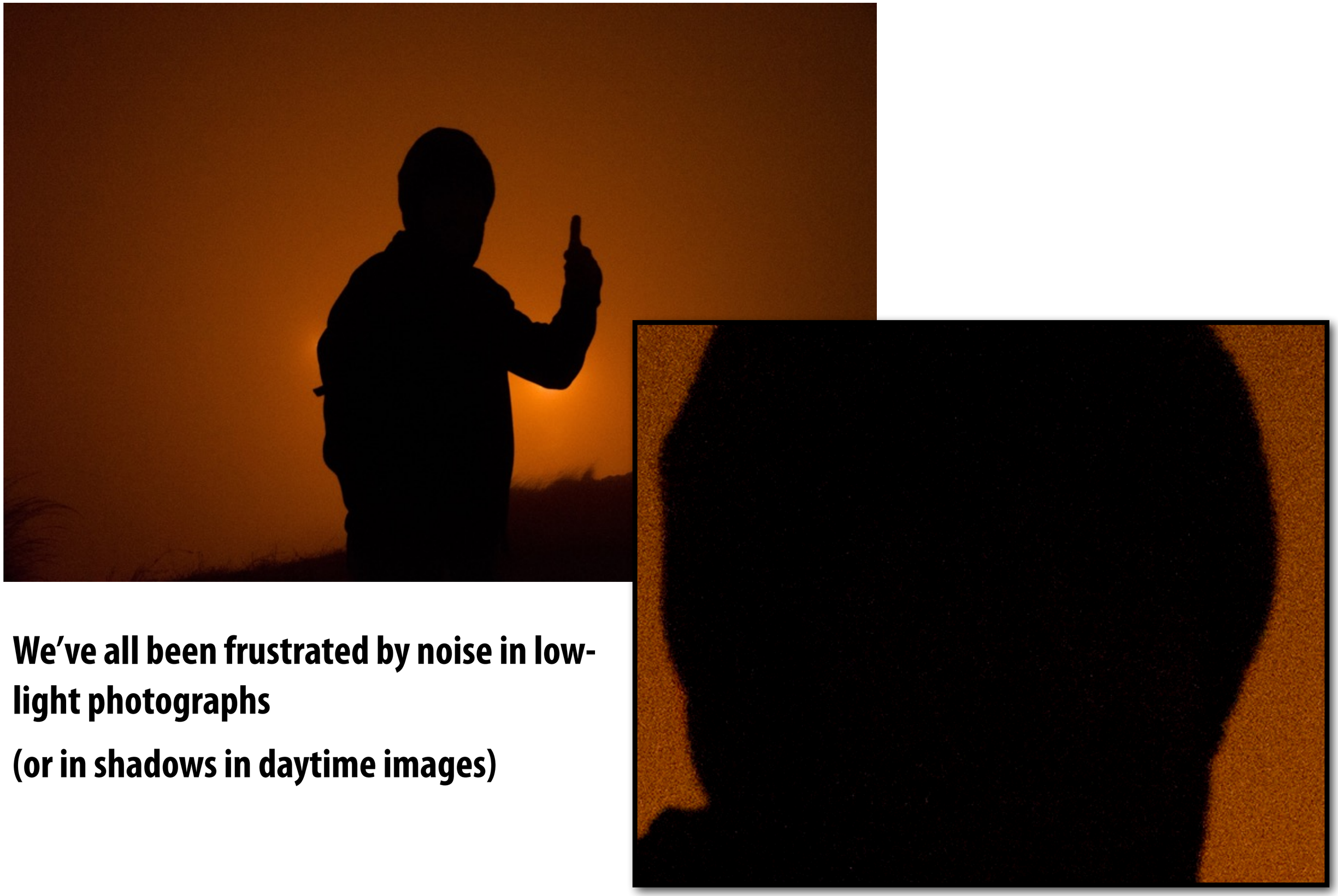


Image credit: Point Grey Research

Measurement noise



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

(or in shadows in daytime images)

Measurement noise

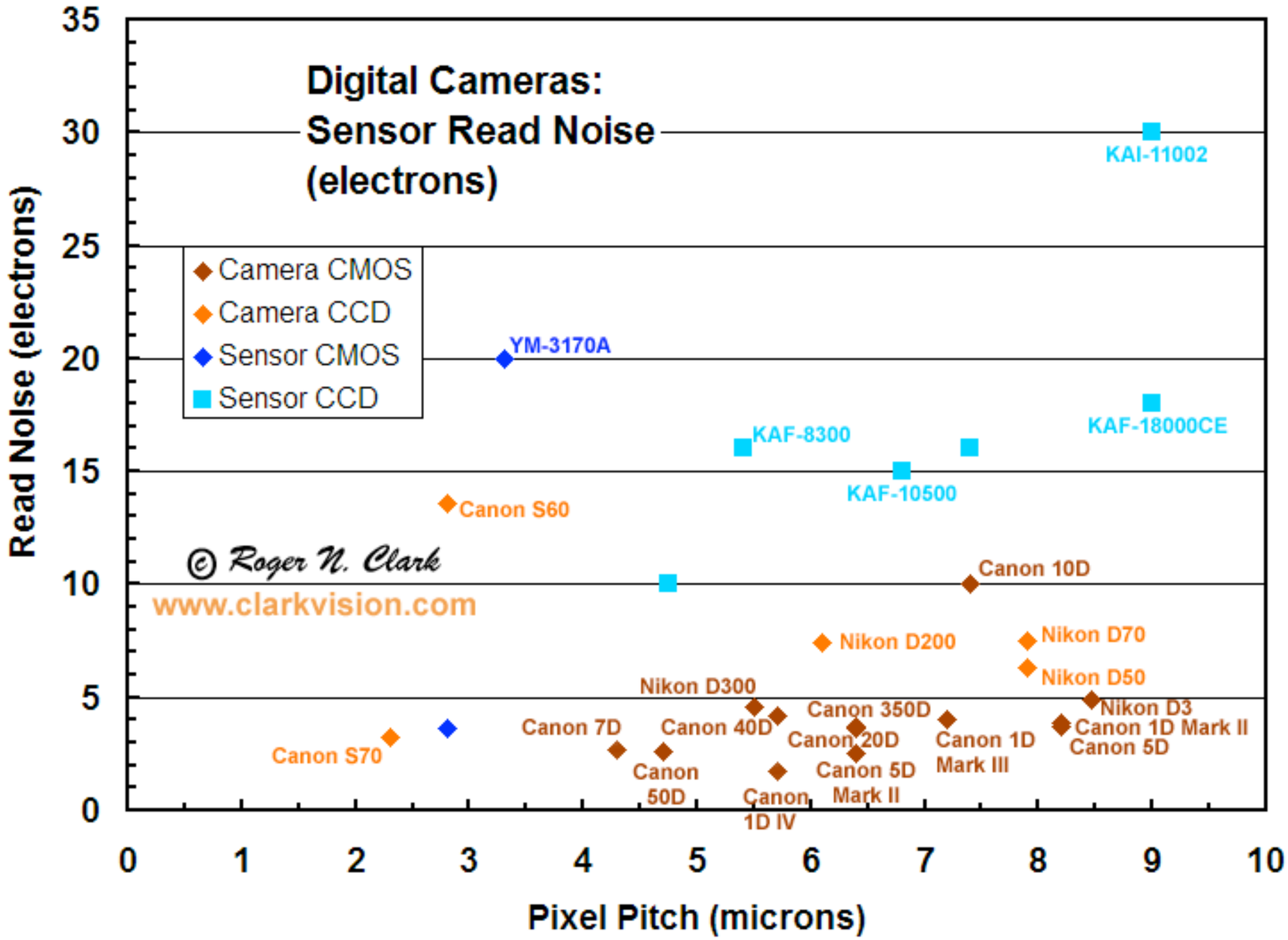
- Photon shot noise:
  - Photon arrival rates feature poisson distribution
  - Standard deviation = sqrt(N)
  - Signal-to-noise ratio (SNR): N/sqrt(N)
  - Brighter the signal the higher the SNR
- Dark-shot noise
  - Due to leakage current
  - Electrons dislodged due to thermal activity (increases exponentially with sensor temperature)

Addressed by: subtract dark image

- Non-uniformity of pixel sensitivity (due to manufacturing defects)
- Read noise
  - e.g., due to amplification / ADC

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

Read noise



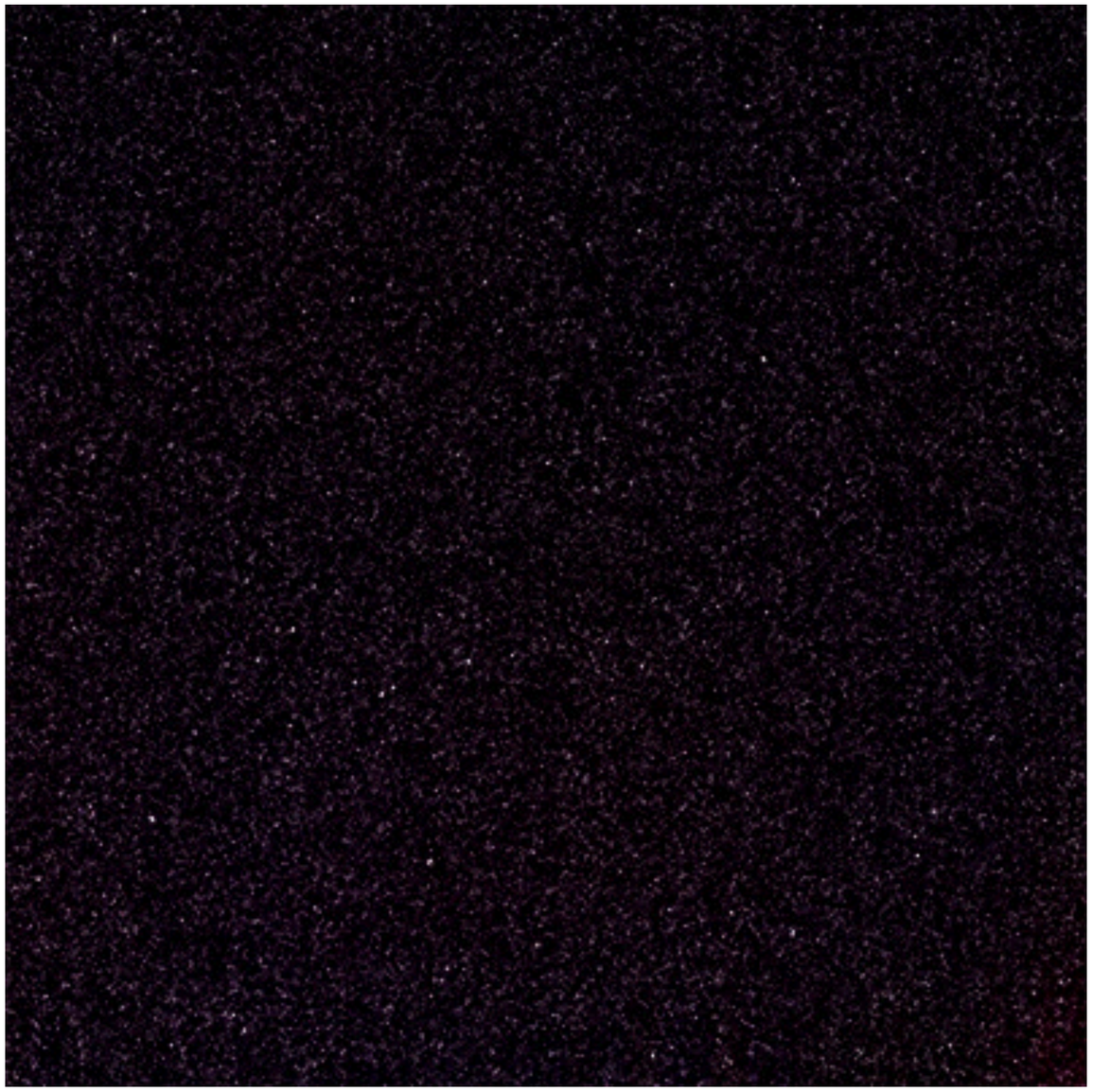
Read noise is largely independent of pixel size

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

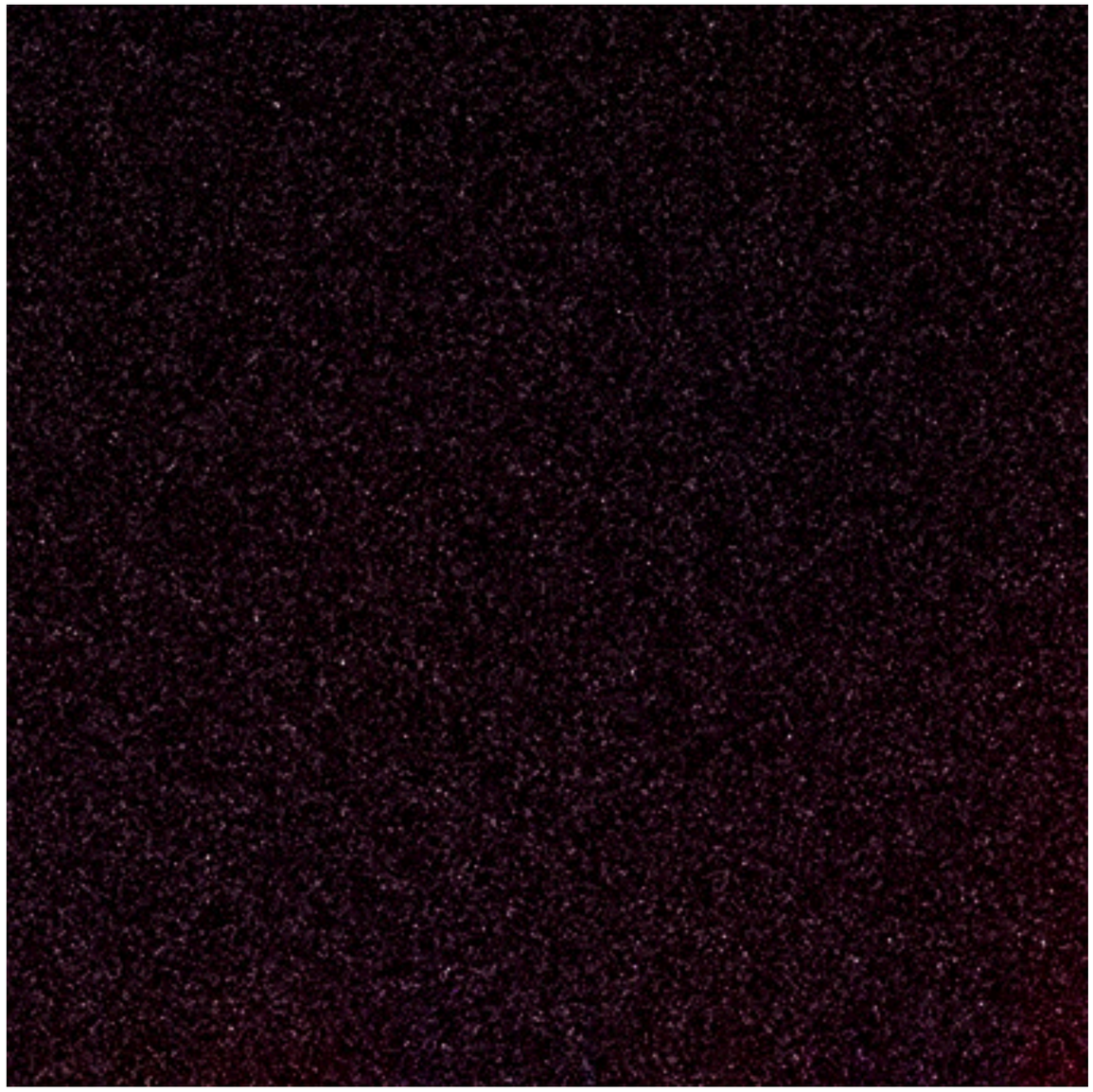
Image credit: clarkvision.com

Dark shot noise / read noise

Black image examples: Nikon D7000, High ISO



1/60 sec exposure



1 sec exposure

Maximize light gathering capability

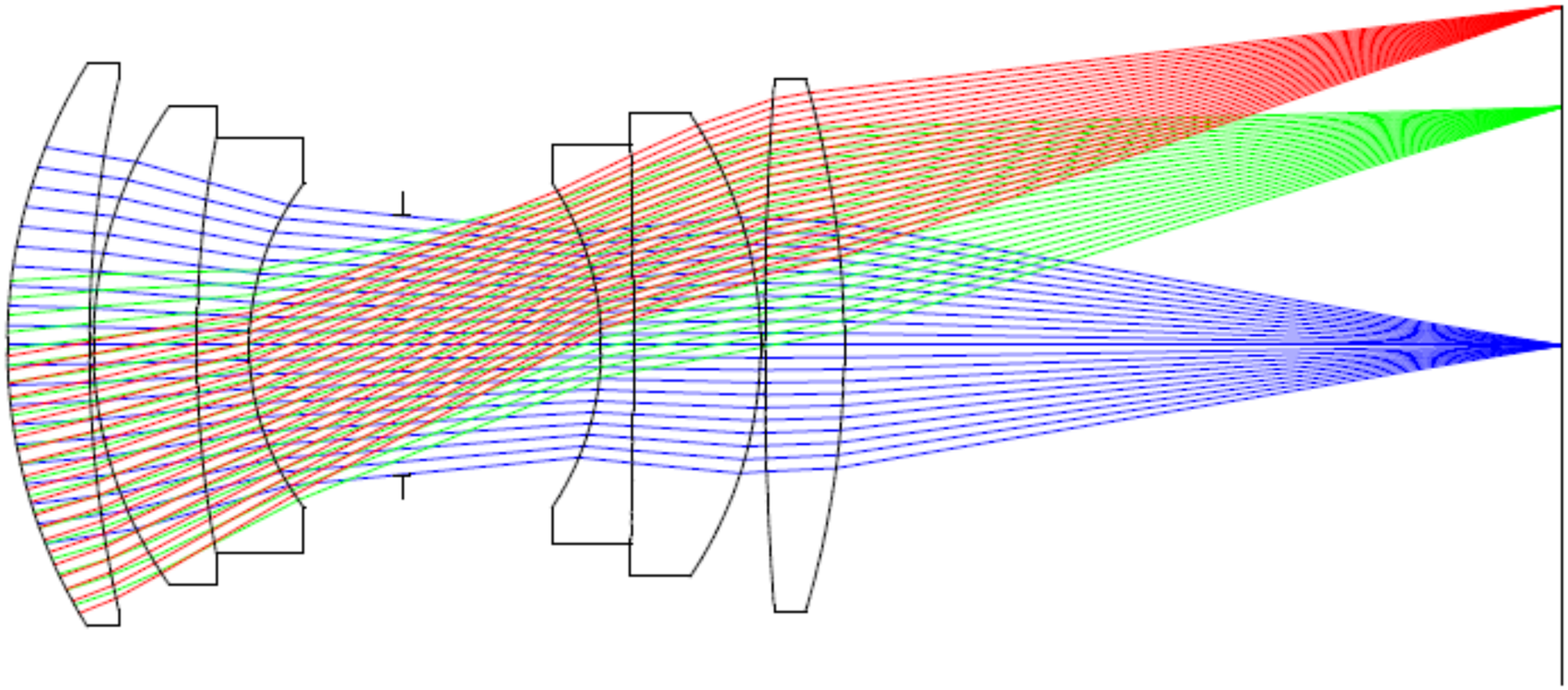
- **Goal: increase signal-to-noise ratio**
  - Dynamic range of a pixel (ratio of brightest light and dimmest light measurable) is determined by the noise floor (minimum signal) and the pixel's full-well capacity (maximum signal)
- **Big pixels**
  - Nikon D4: 7.3 um
  - iPhone 5s: 1.5 um
- **Sensitive 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

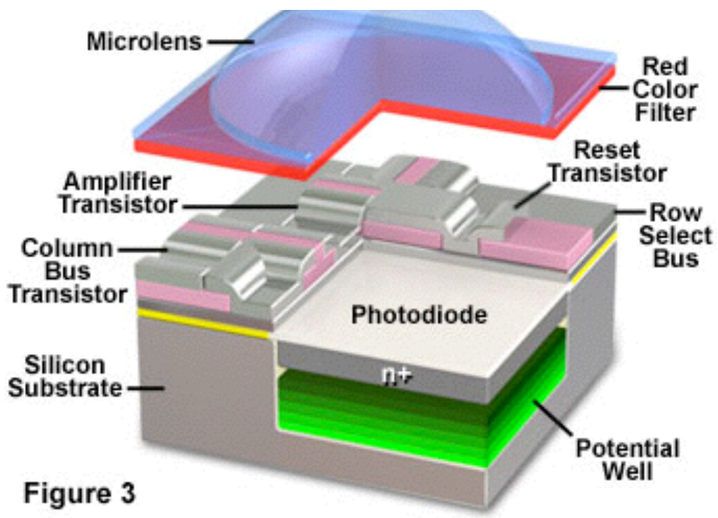
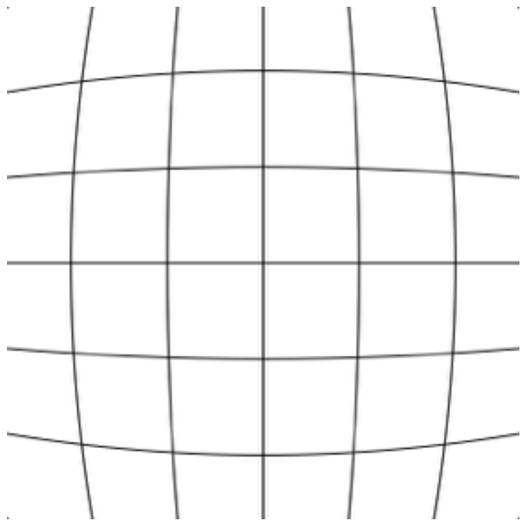


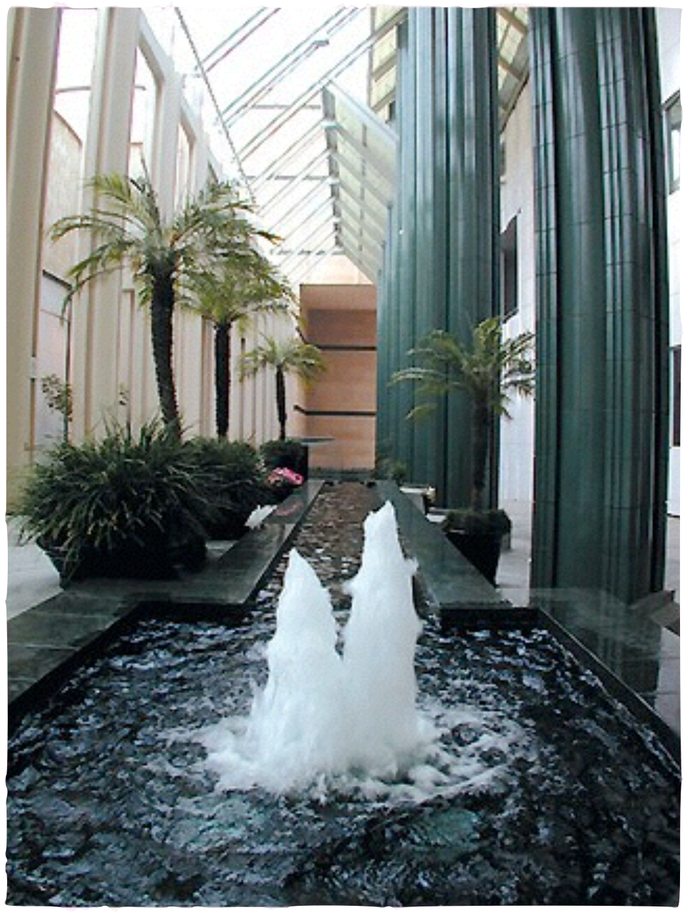
Image credit: Mark Butterworth

More challenges

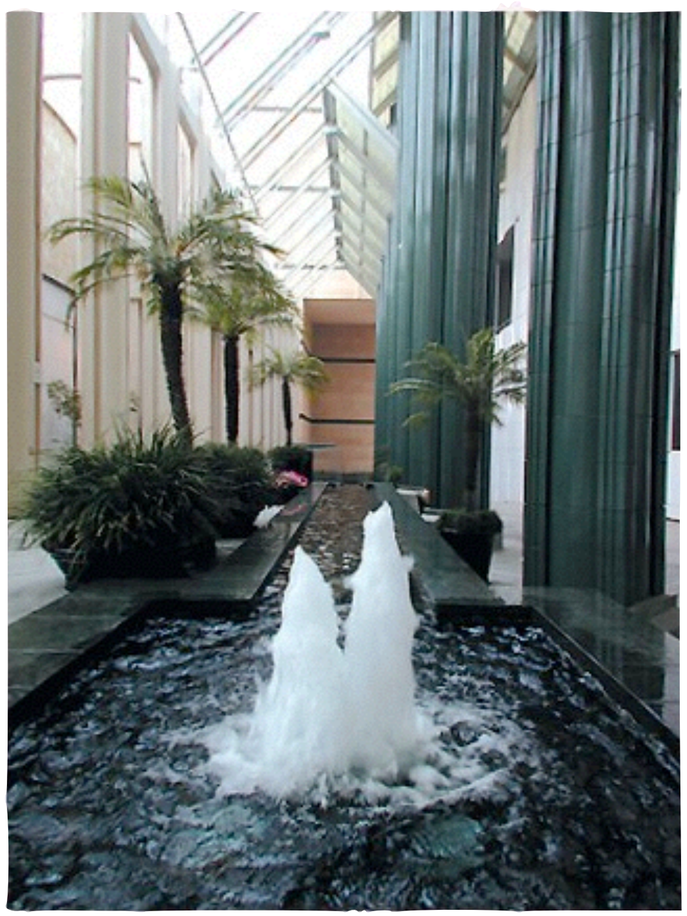
- Chromatic shifts over sensor
  - Pixel light sensitivity changes over sensor due to interaction with microlens (Recall index of refraction depends on wavelength, so some wavelengths are more likely to suffer from cross-talk or reflection. Ugg!)
- Dead pixels (stuck at white or black)
- Lens distortion



Pincushion distortion



Captured Image



Corrected Image

Image credit: PCWorld

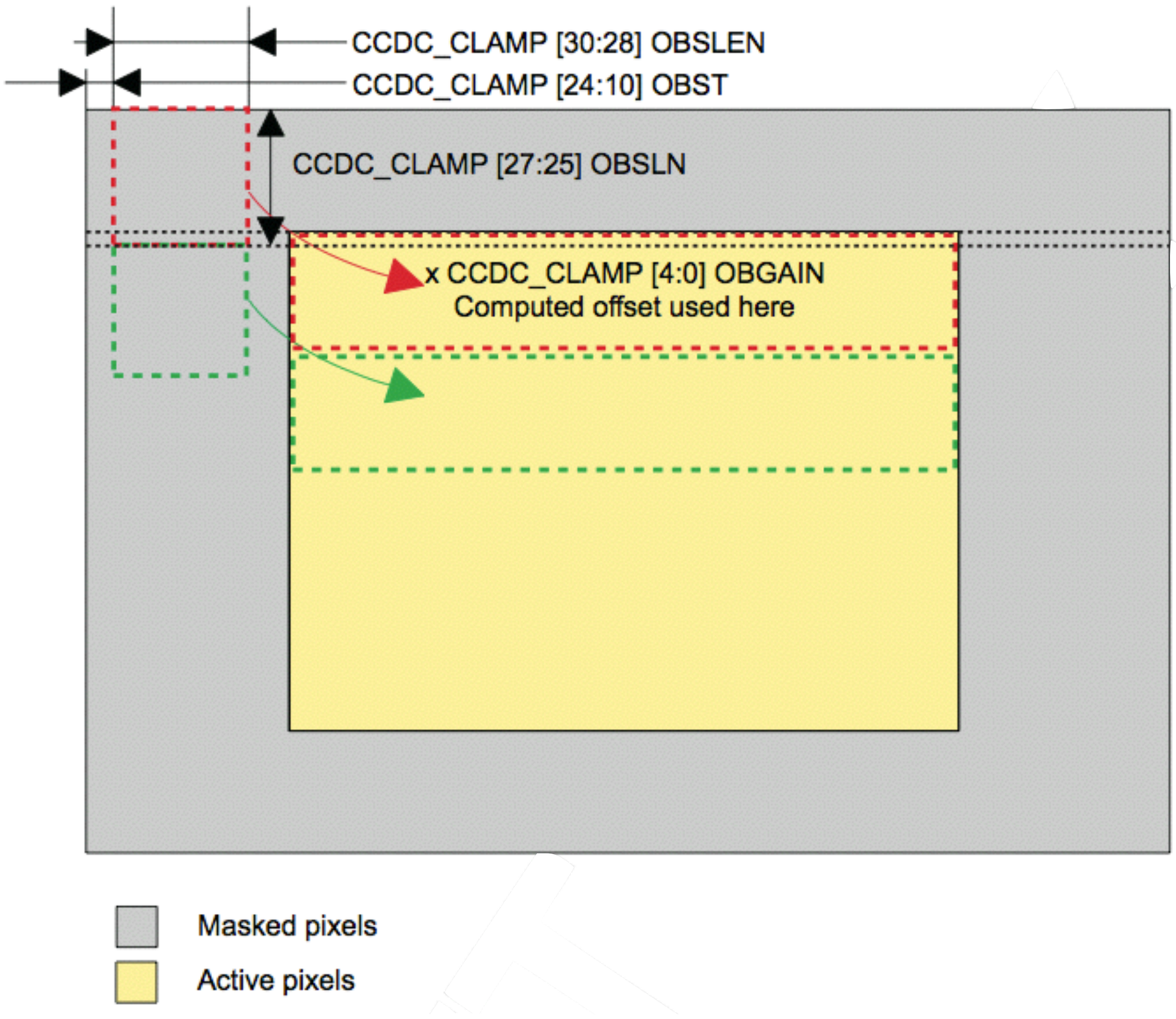
A simple RAW image processing pipeline

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]
```



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 2D buffer of 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];
```

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)
```

- Determine white balance coefficients 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

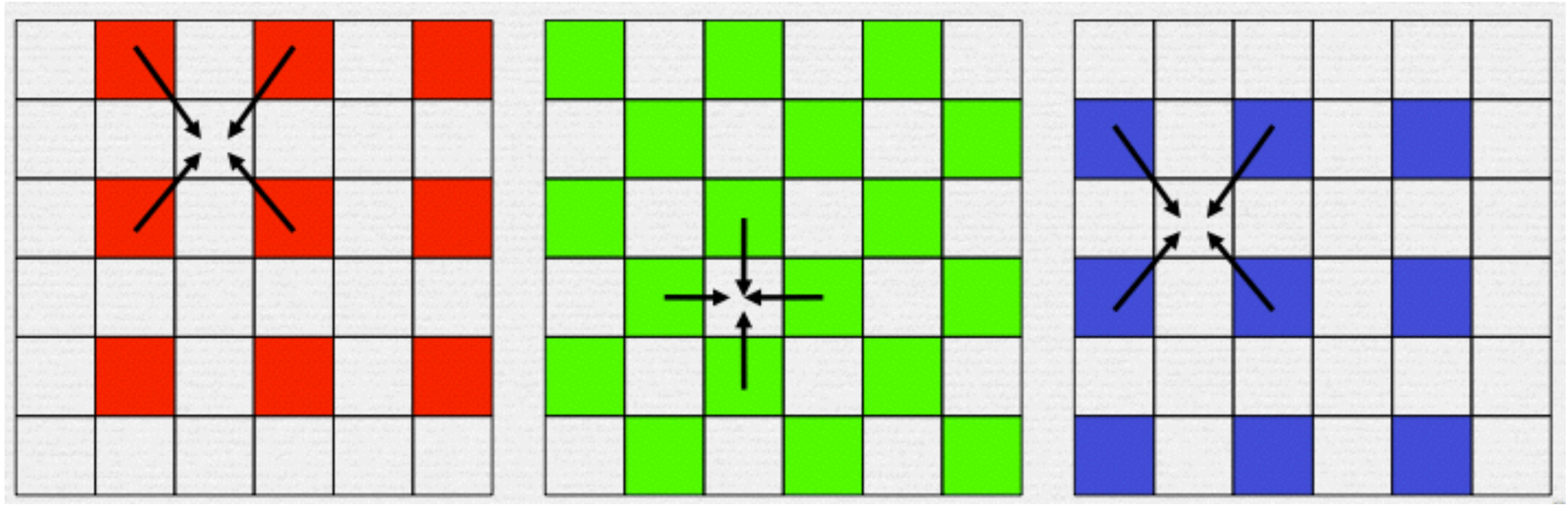
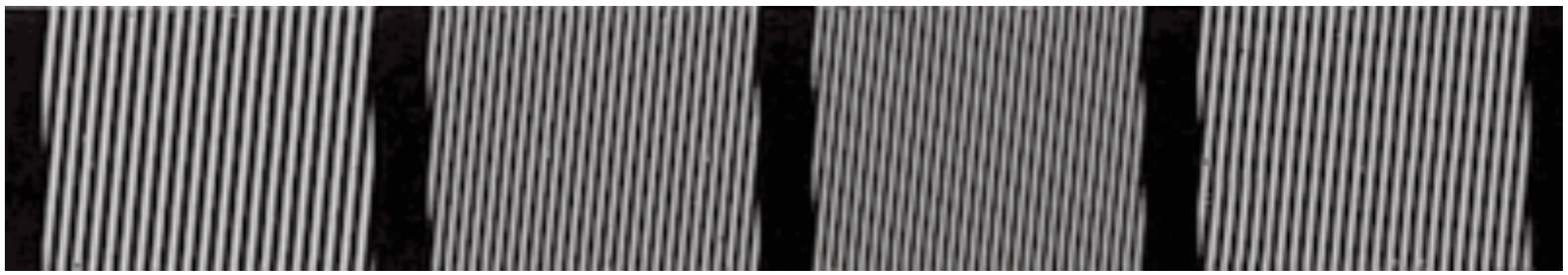


Image credit: Mark Levoy

Demosaicing errors

- Moire pattern color artifacts
  - Common trigger: fine diagonal black and white stripes
  - Common solution:
    - Convert demosaiced value to YCbCr
    - Low-pass filter CbCr channels
    - Combine prefiltered CbCr with full resolution Y from sensor to get RGB

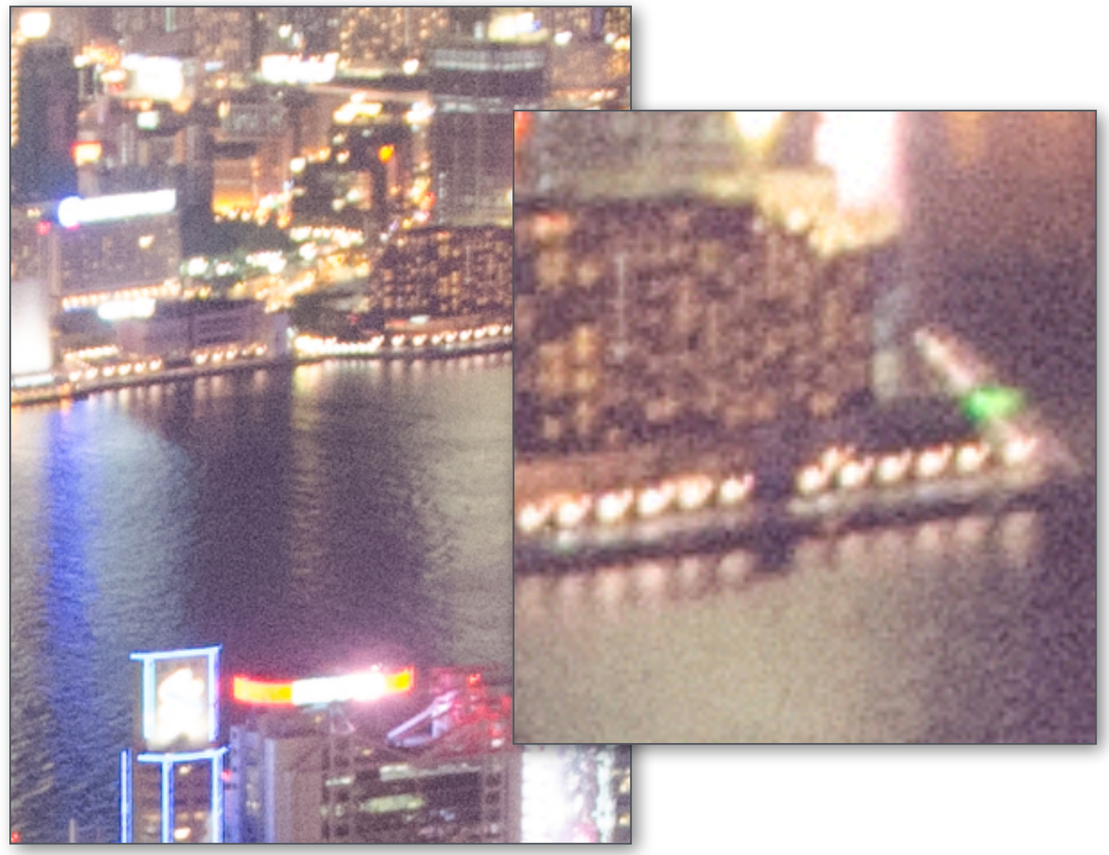


RAW data from sensor

Demosaiced

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

Denoising: effect of downsizing on image noise

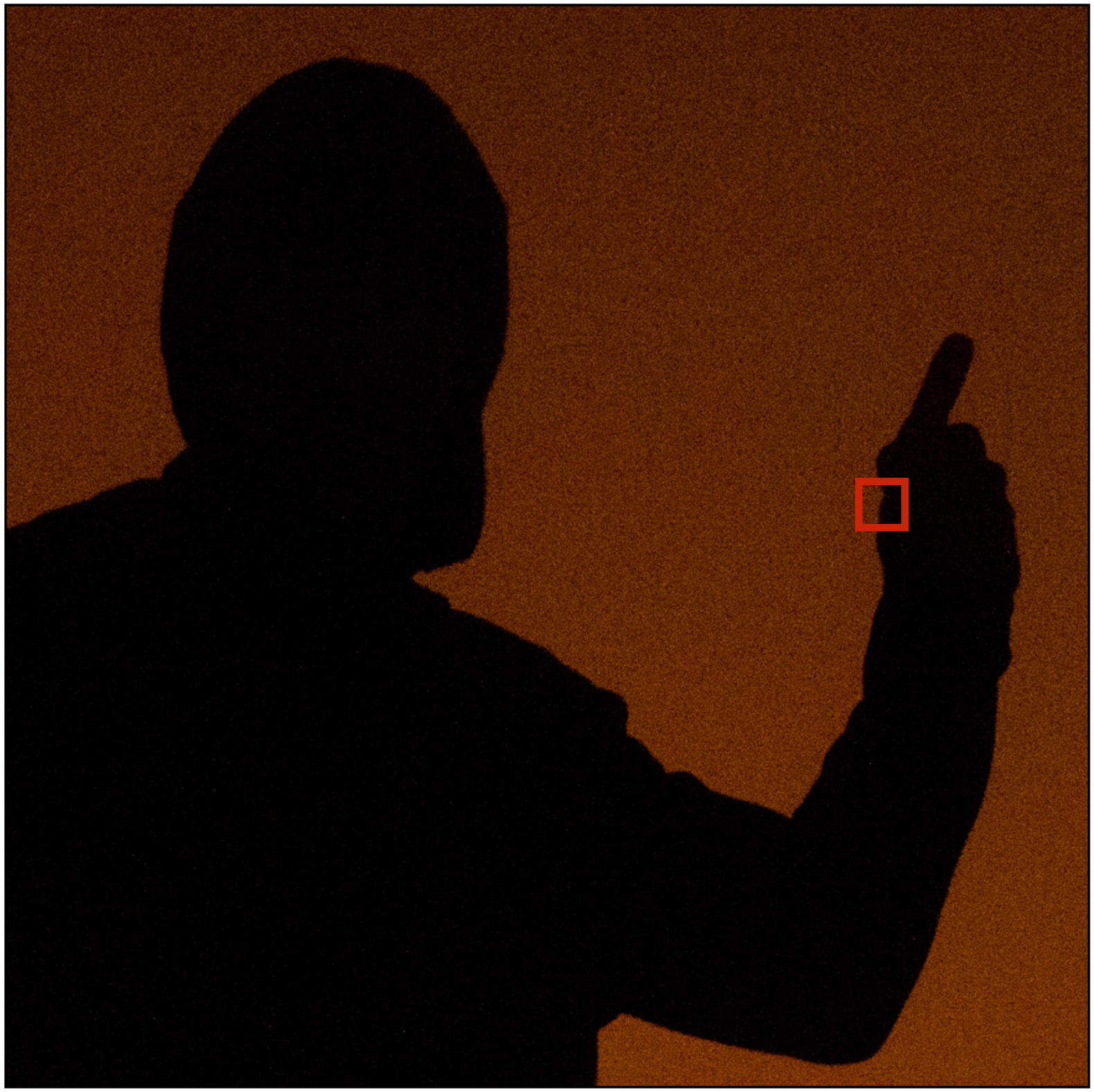


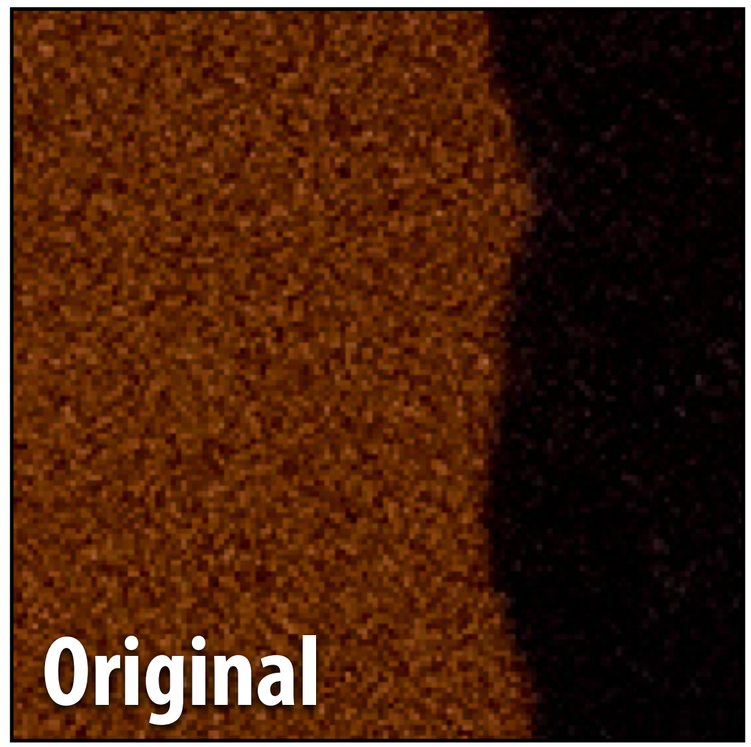
point sampled

averaged down (bilinear resampling)

Credit: Levoy

Denoising





Aside: image processing basics

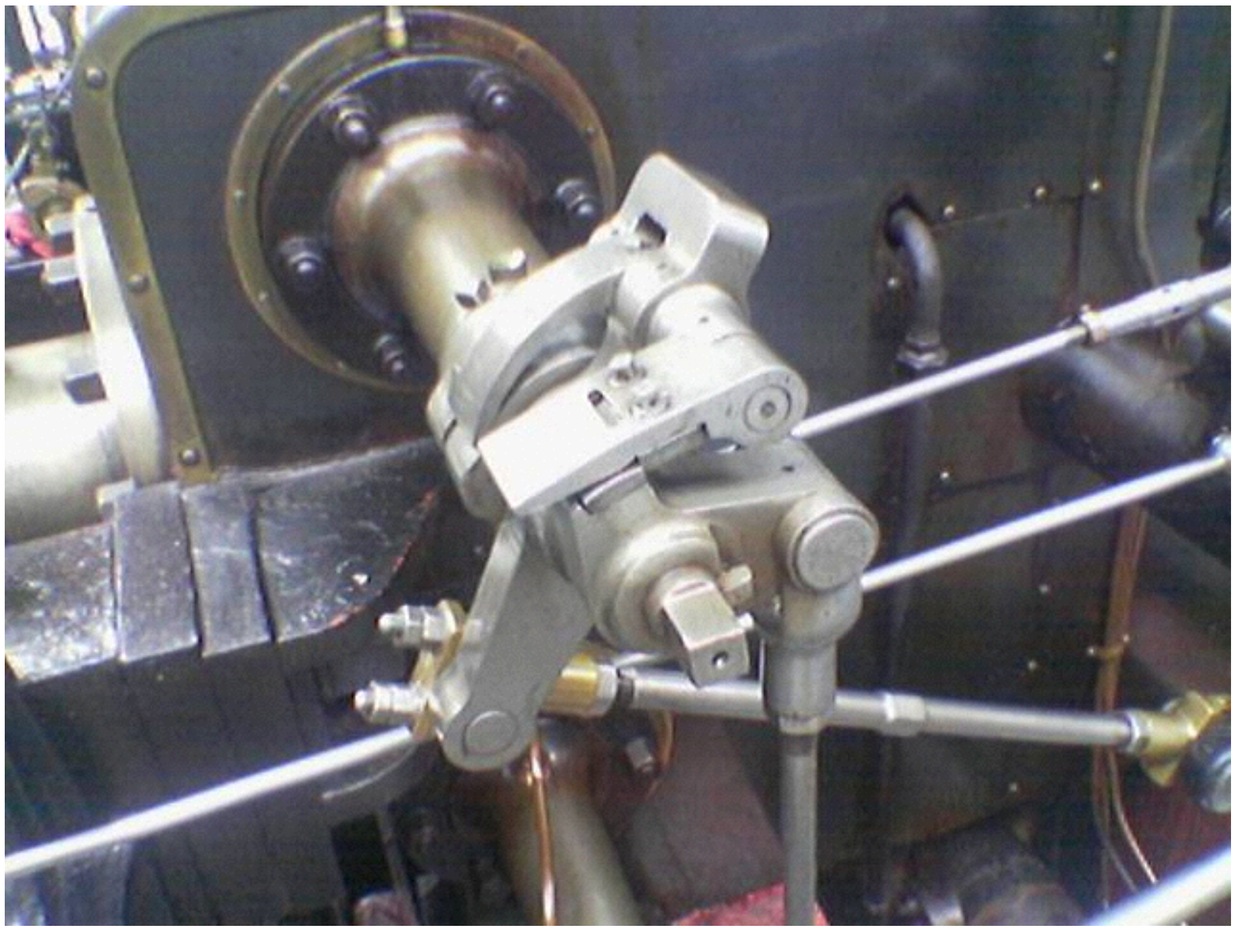
Example image processing operations

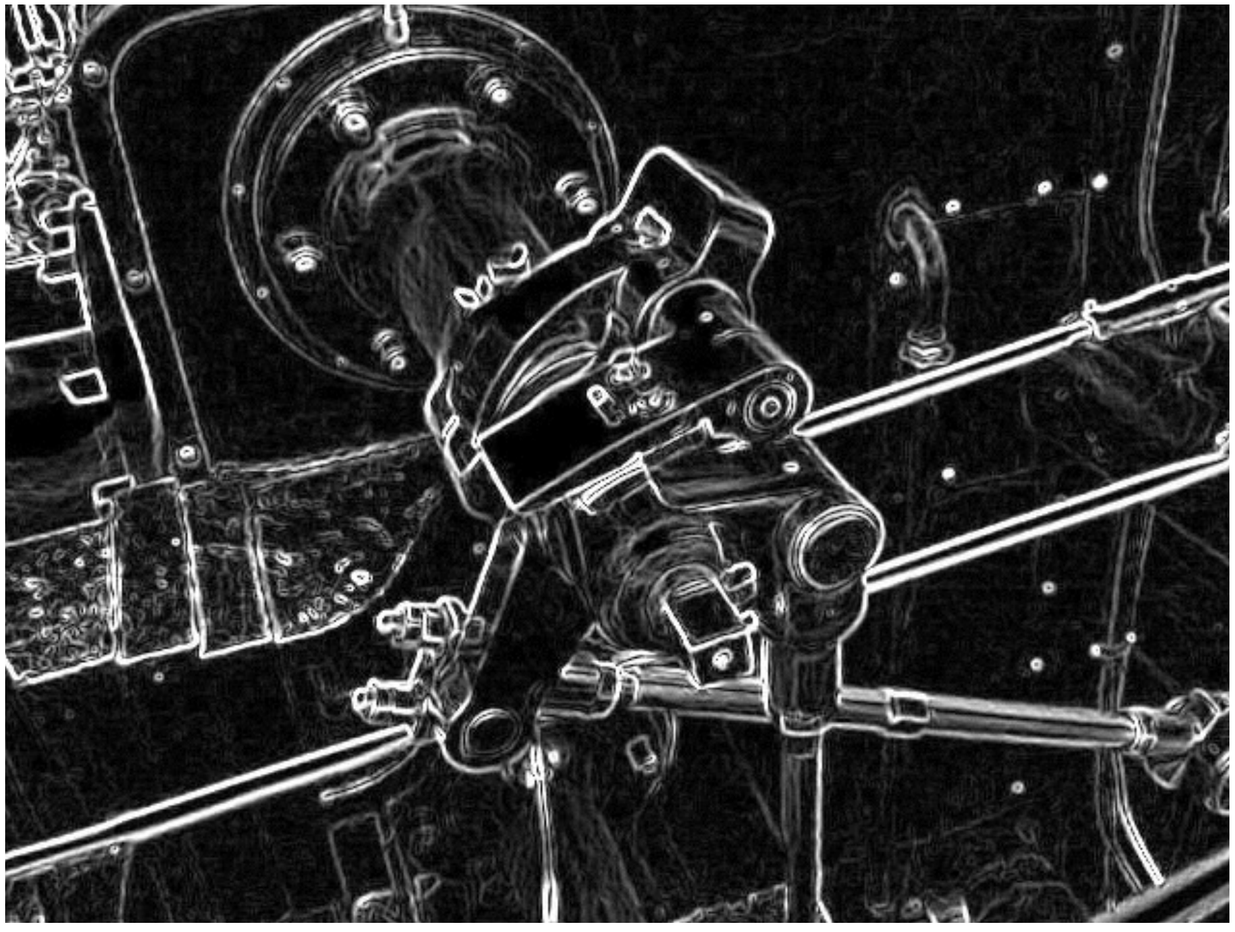
Blur

Example image processing operations

Sharpen

Edge detection





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

Review: convolution

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

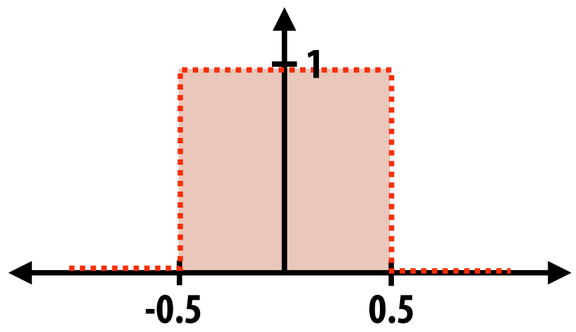
output signal

filter

input signal

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 "smoothed" version of *g*

Discrete 2D convolution

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

output image | filter | 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:

$$f(i, j) = \mathbf{F}_{i,j}$$

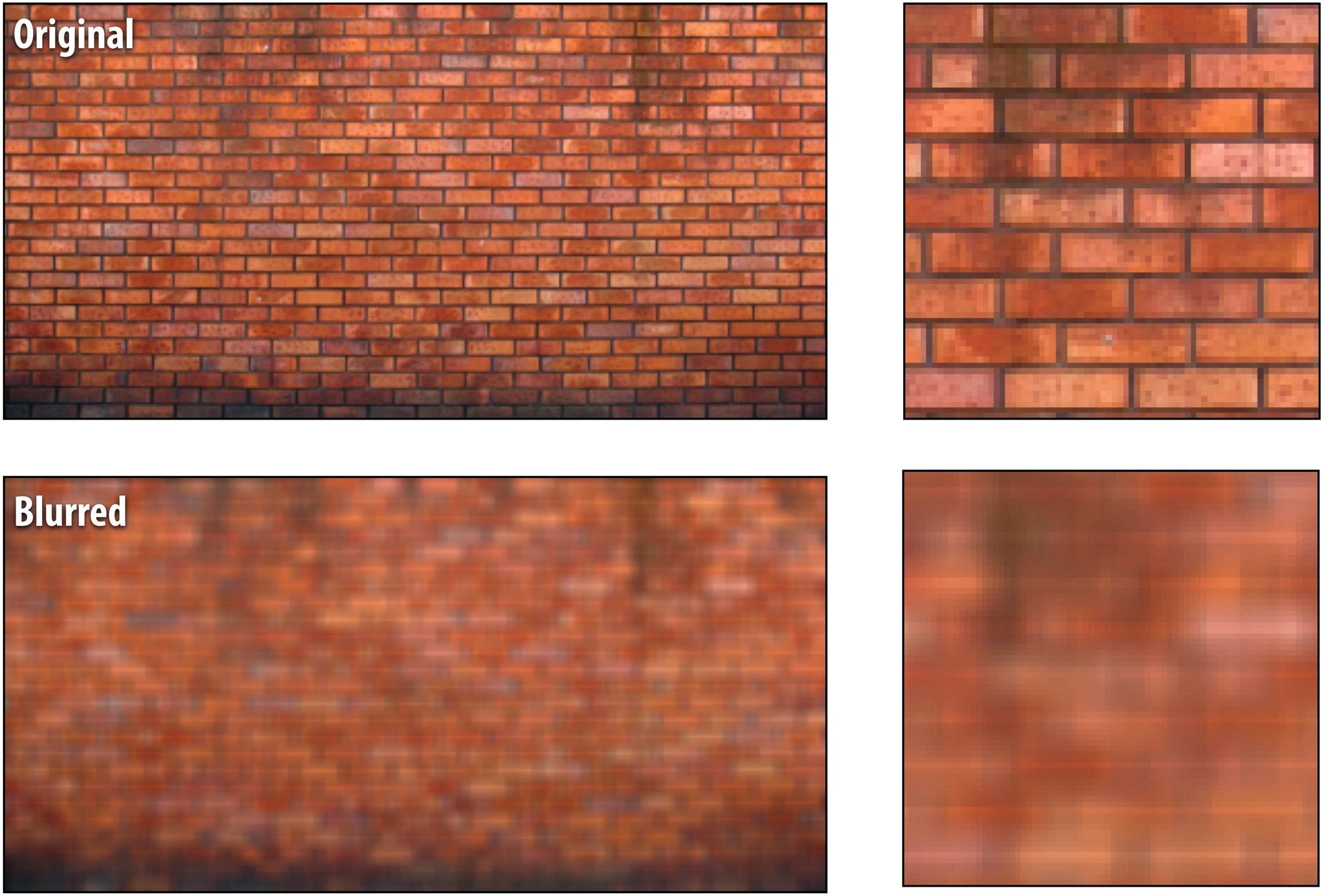
(often called: "filter weights", "kernel")

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

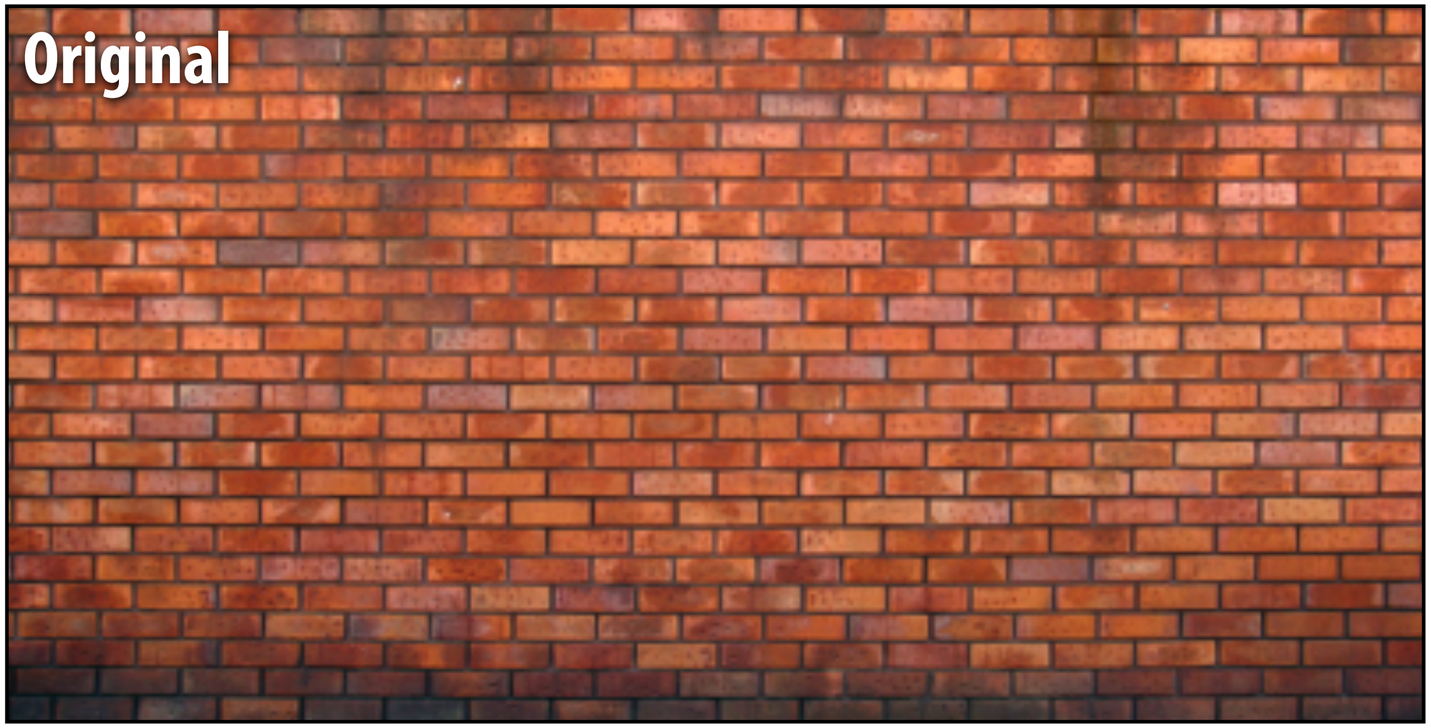
- **Obtain filter coefficients from sampling 2D Gaussian**

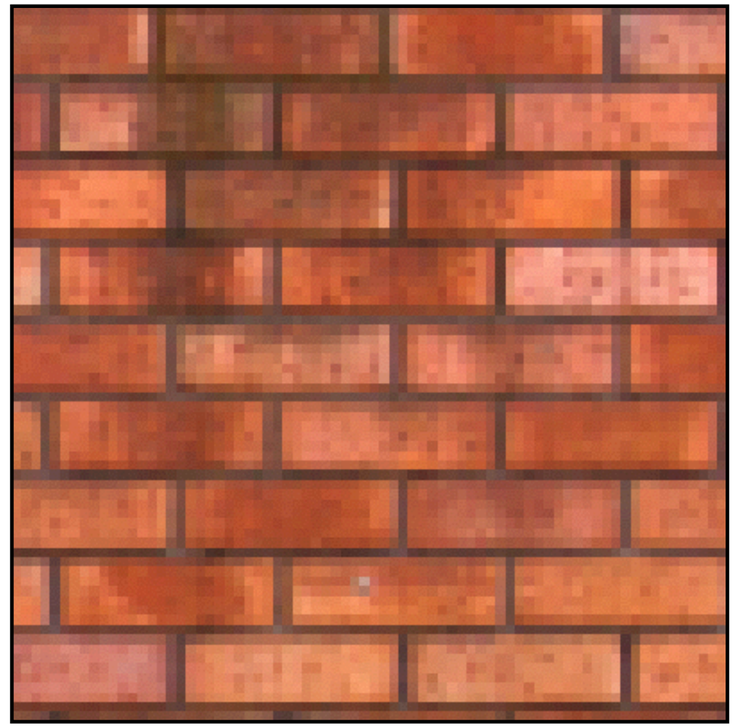
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

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

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

7x7 gaussian blur



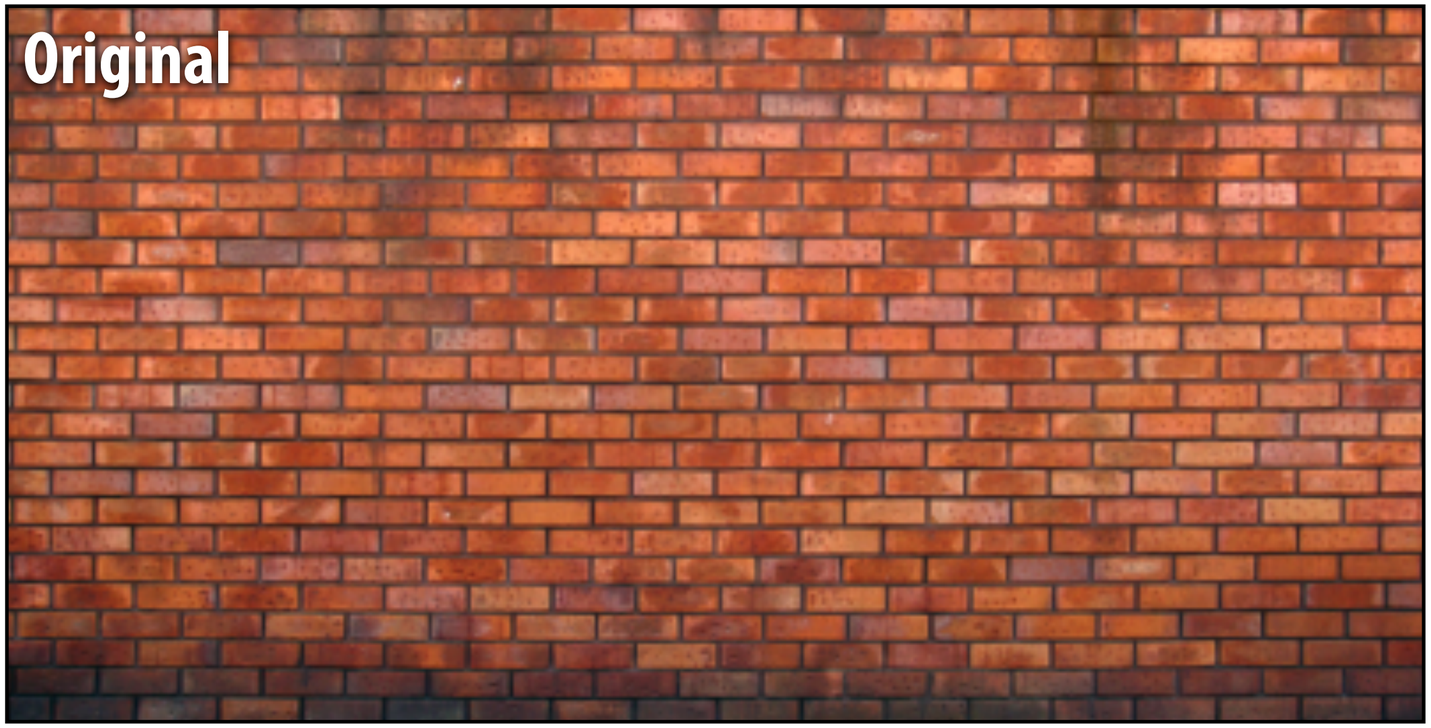


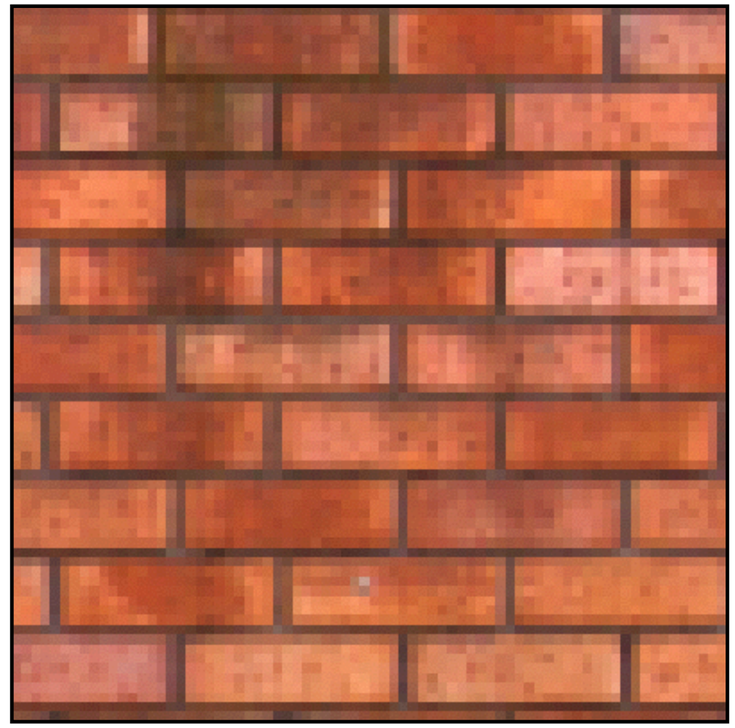
What does convolution with this filter do?

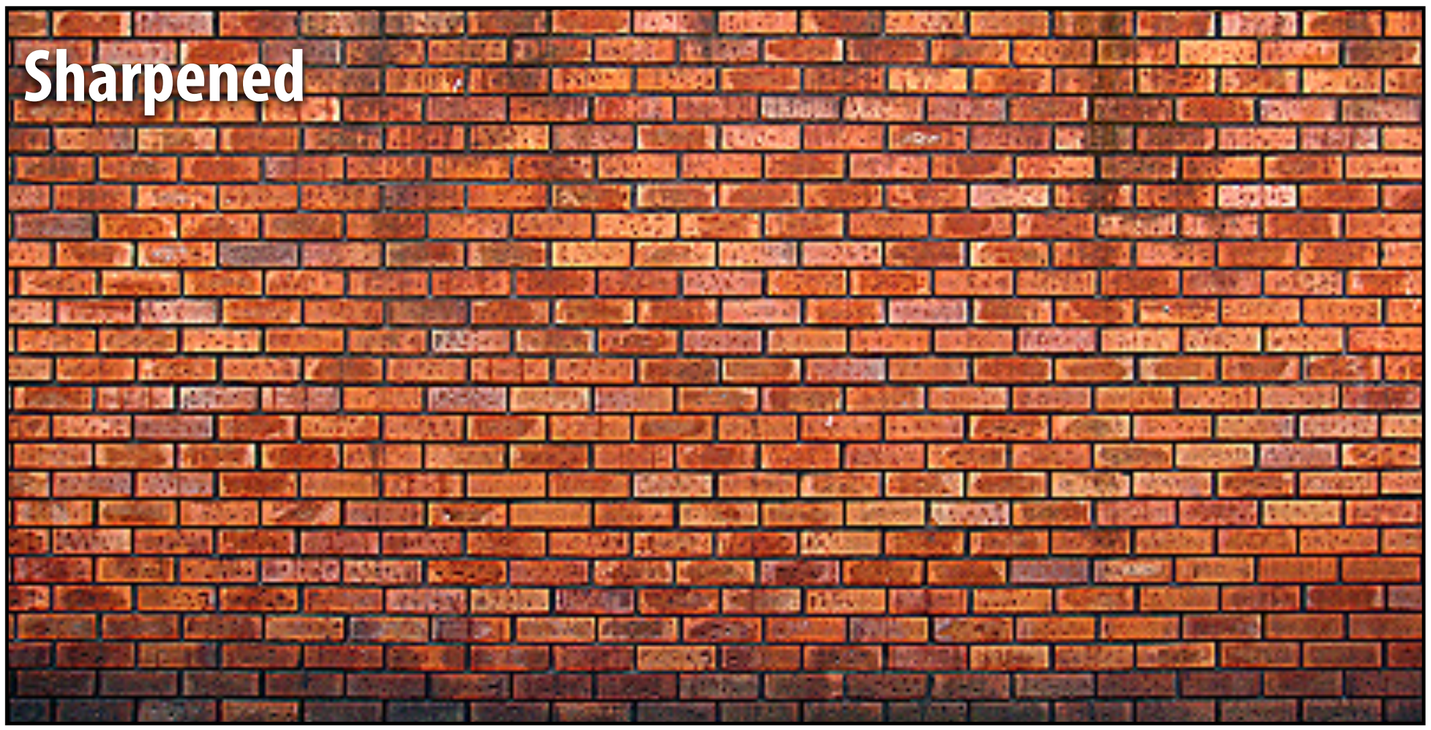
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Sharpens image!

3x3 sharpen filter







What does convolution with these filters do?

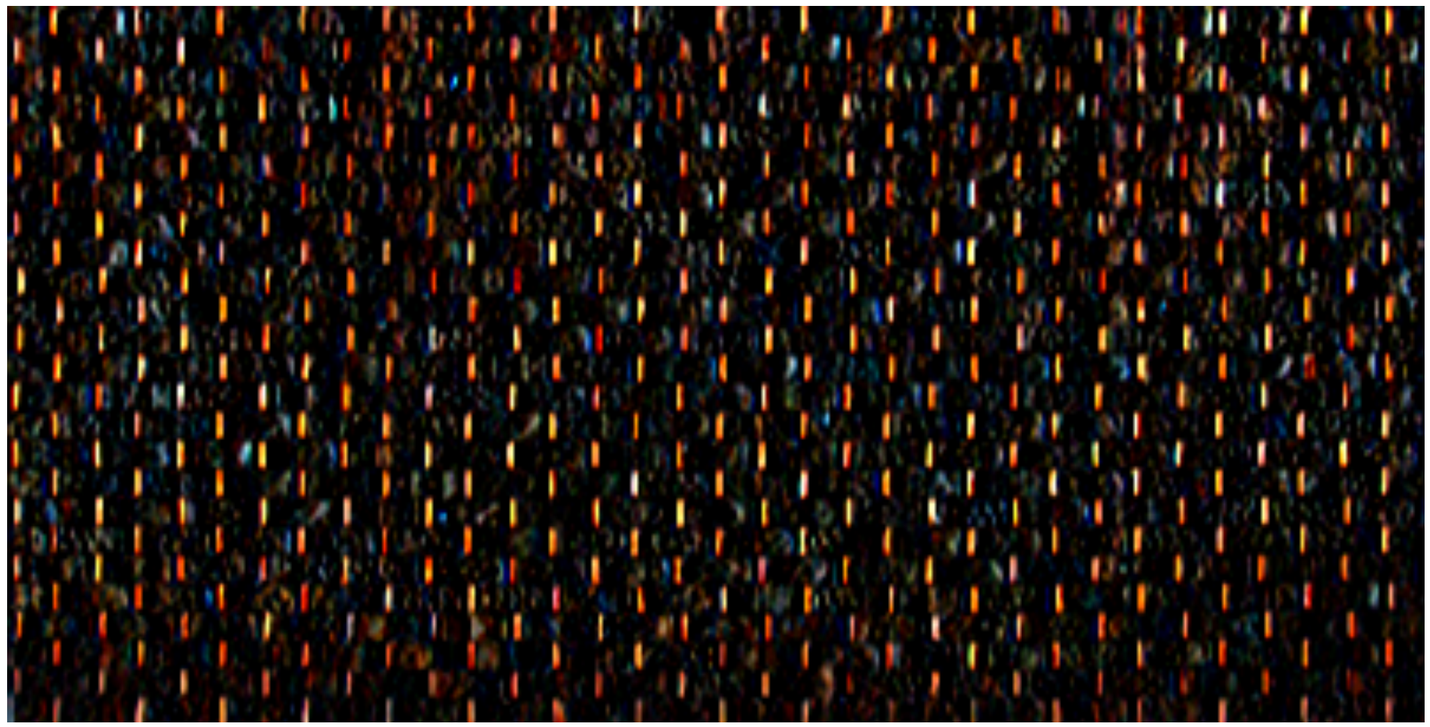
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

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

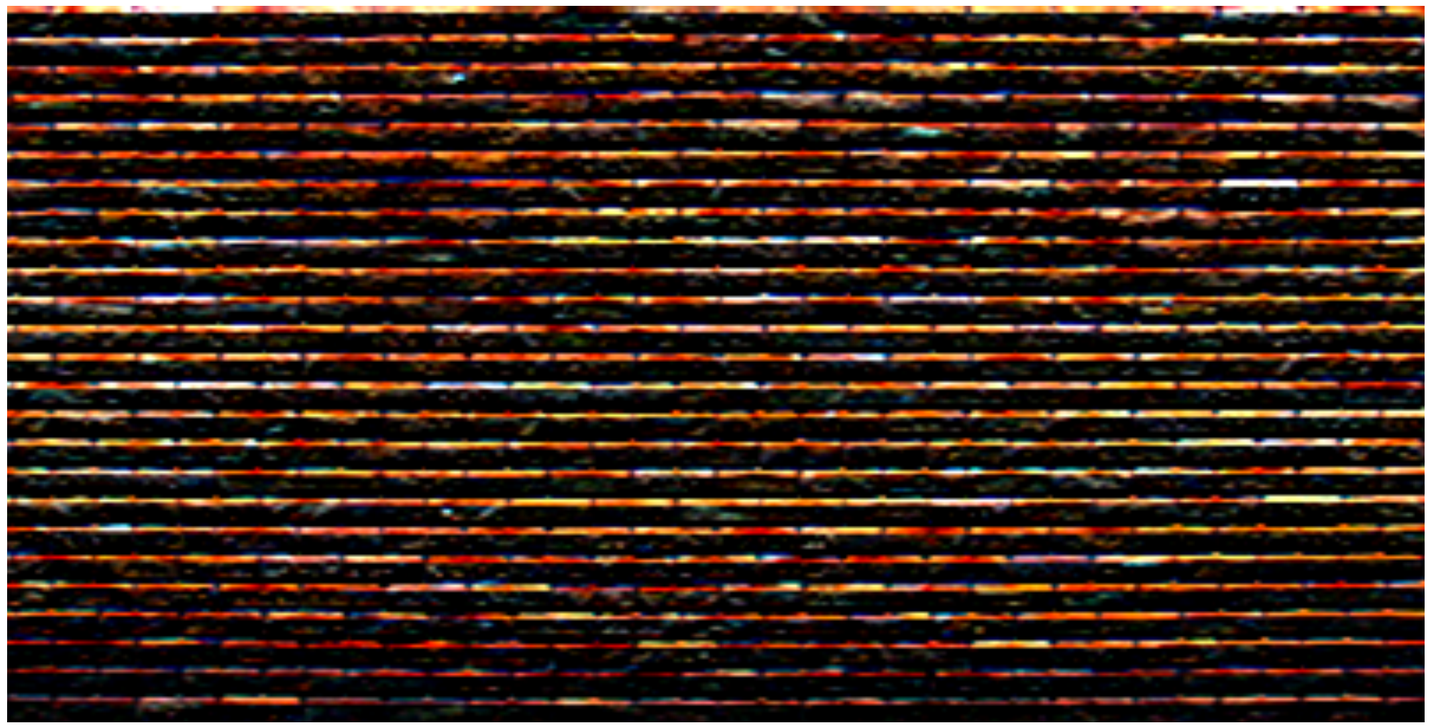
Extracts horizontal gradients

Extracts vertical gradients

Gradient detection filters



Horizontal gradients



Vertical gradients

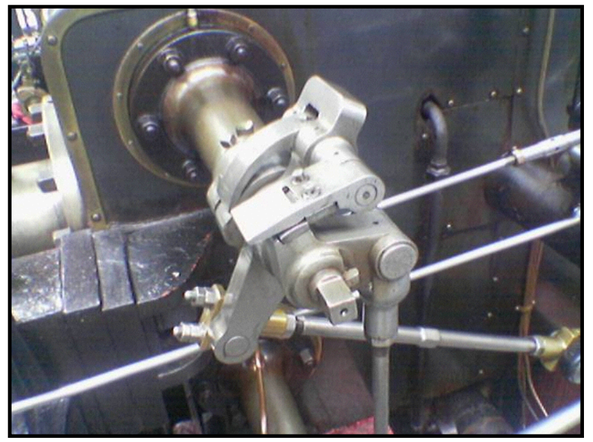
Note: you can think of a filter as a "detector" of a pattern, and the magnitude of a pixel in the output image as the "response" of the filter to the region surrounding each pixel in the input image (this is a common interpretation in computer vision)

Sobel edge detection

- **Compute gradient response images**

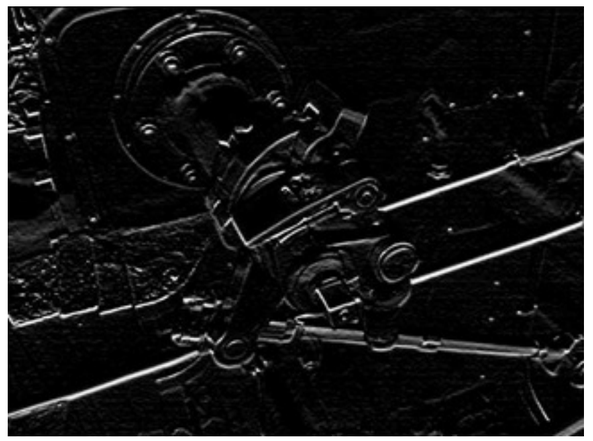
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I$$

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



G_x

G_y

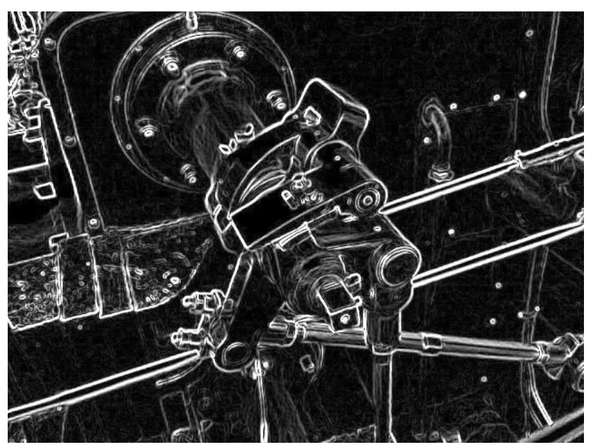


- **Find pixels with large gradients**

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

Pixel-wise operation on images

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 is the 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)

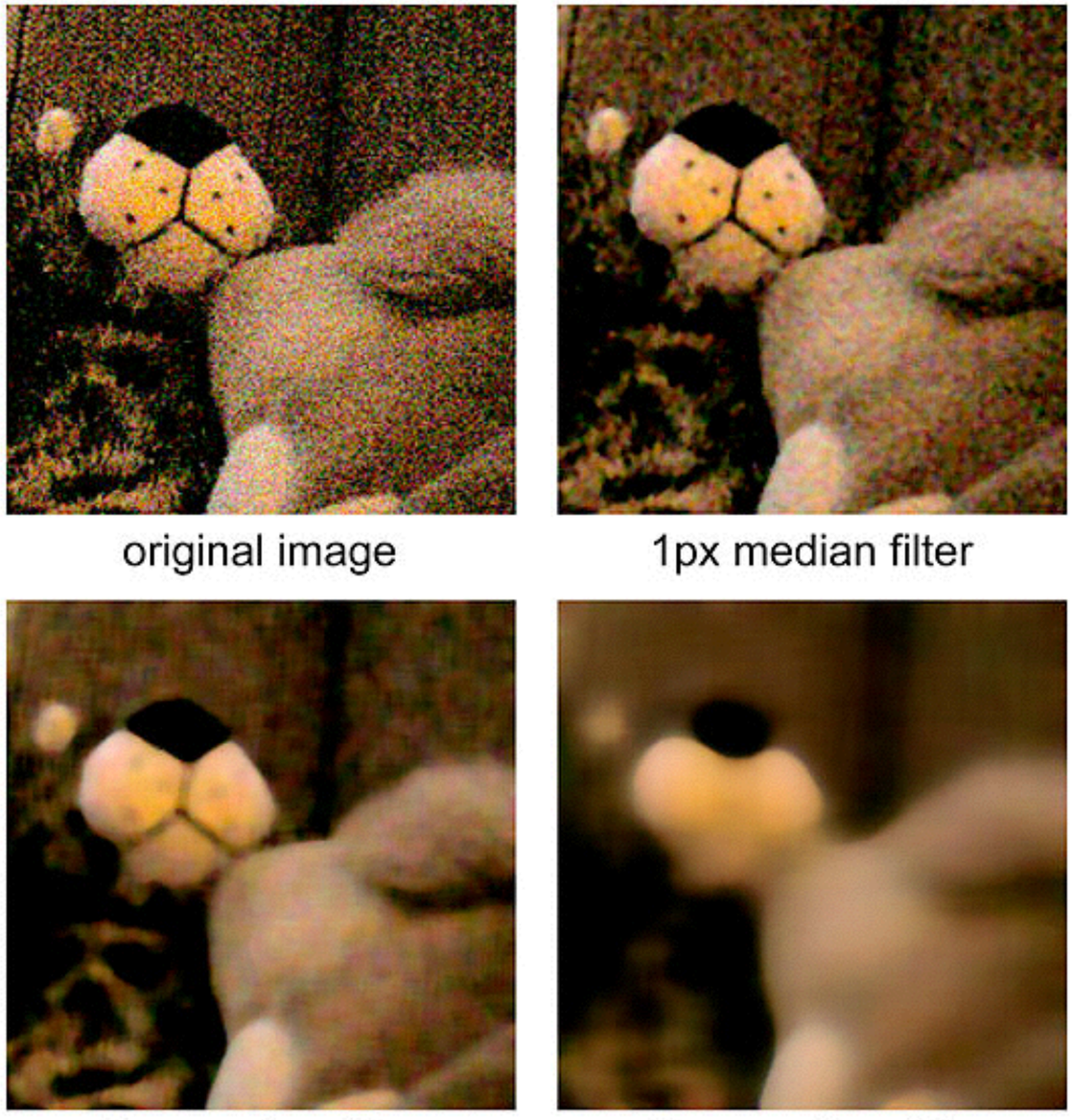
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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
   }
}
```



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

Bilateral filter

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

Bilateral filter

$$\mathrm{BF}[I](x,y) = \sum_{i,j} f(\|I(x-i,y-j) - I(x,y)\|) G(i,j) I(x-i,y-j)$$

Gaussian blur kernel

Input image

For all pixels in support region of Gaussian kernel

Re-weight based on difference in input image pixel values

- 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: $f(x) = 0 \; if \; x > threshold, \; 1 \; otherwise$

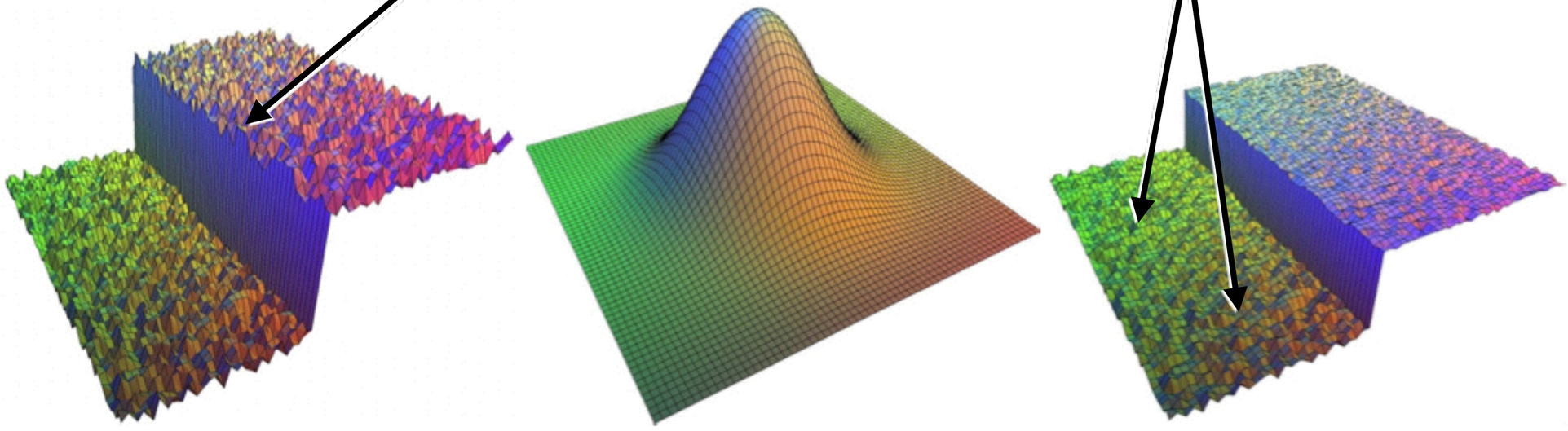
Value of output pixel (x,y) is the weighted sum of all pixels in the support region of a truncated gaussian kernel

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*

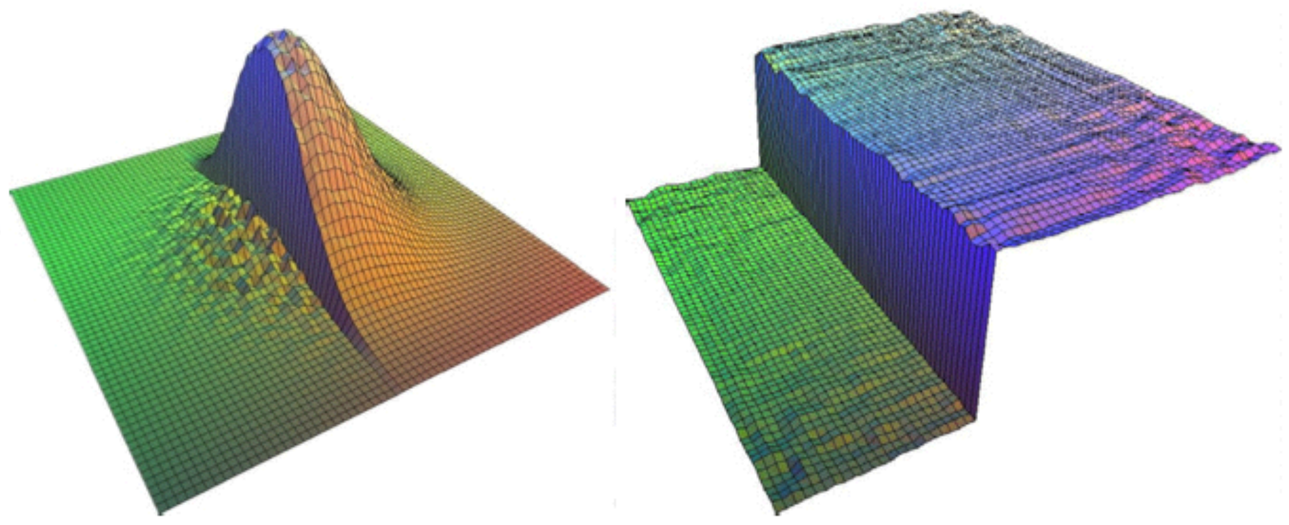
Pixels with significantly different intensity as *p* contribute little to filtered result (they are "on the "other side of the edge"



Input image

G(): gaussian about input pixel *p*

f(): Influence of support region

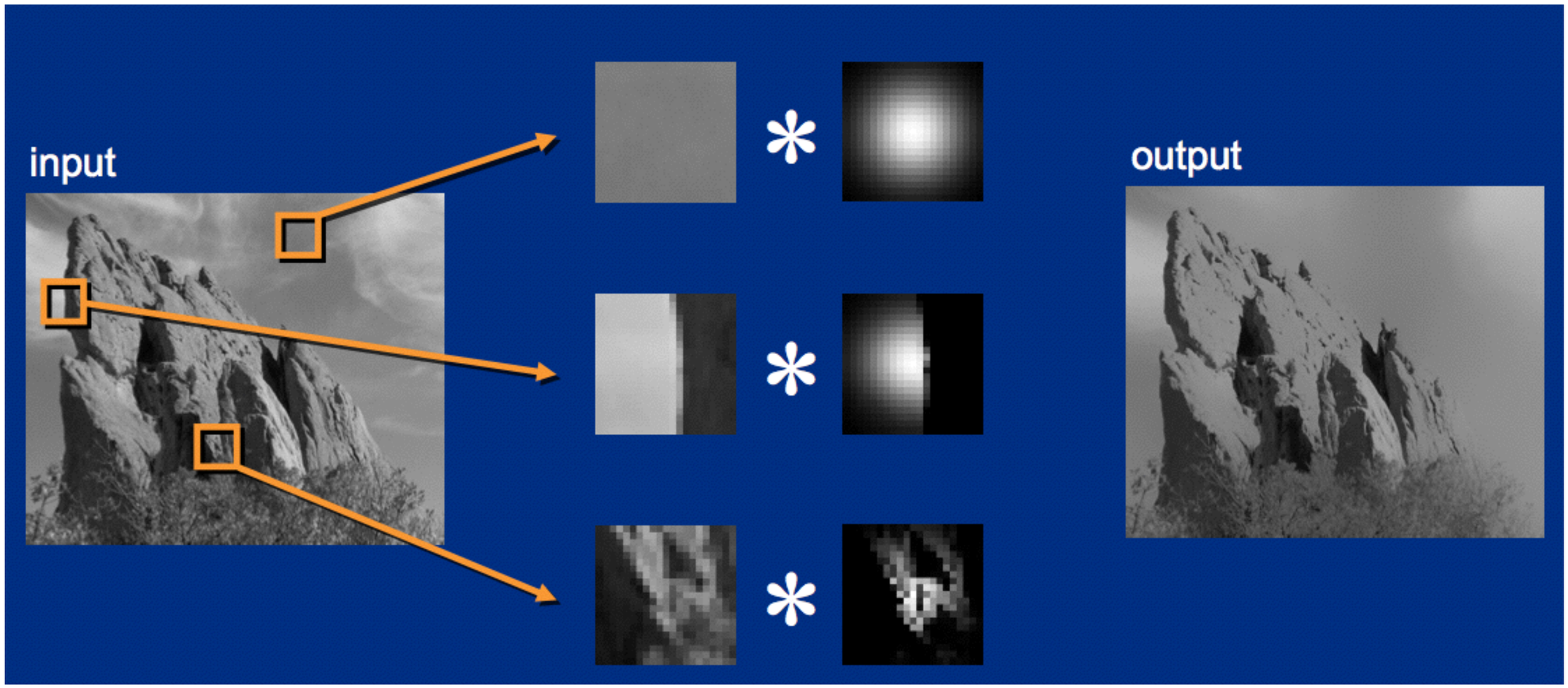


G x f: filter weights for pixel *p*

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.

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

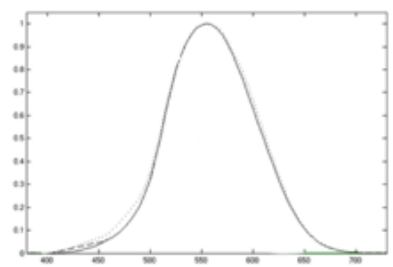
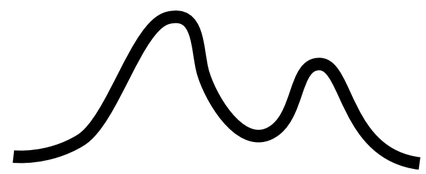
```
output_rgb_pixel = COLOR_CONVERSION_MATRIX * input_rgb_pixel
```

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

Represented in sensor-specific basis

Note: modern pipelines will 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)

Lightness (perceived brightness) aka luma

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

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

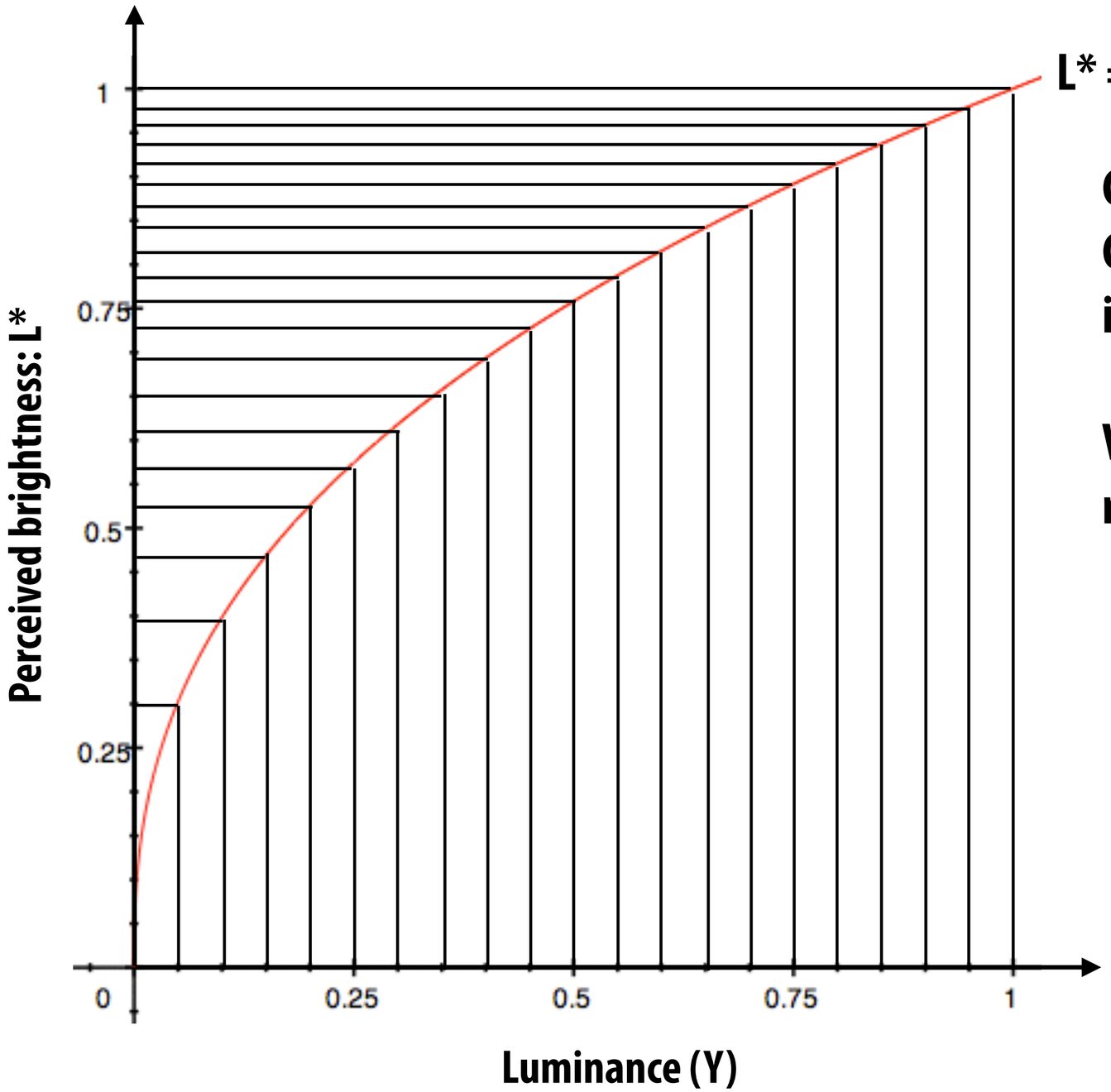
In a dark room, you turn on a light with luminance: Y_1

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 encode luminance



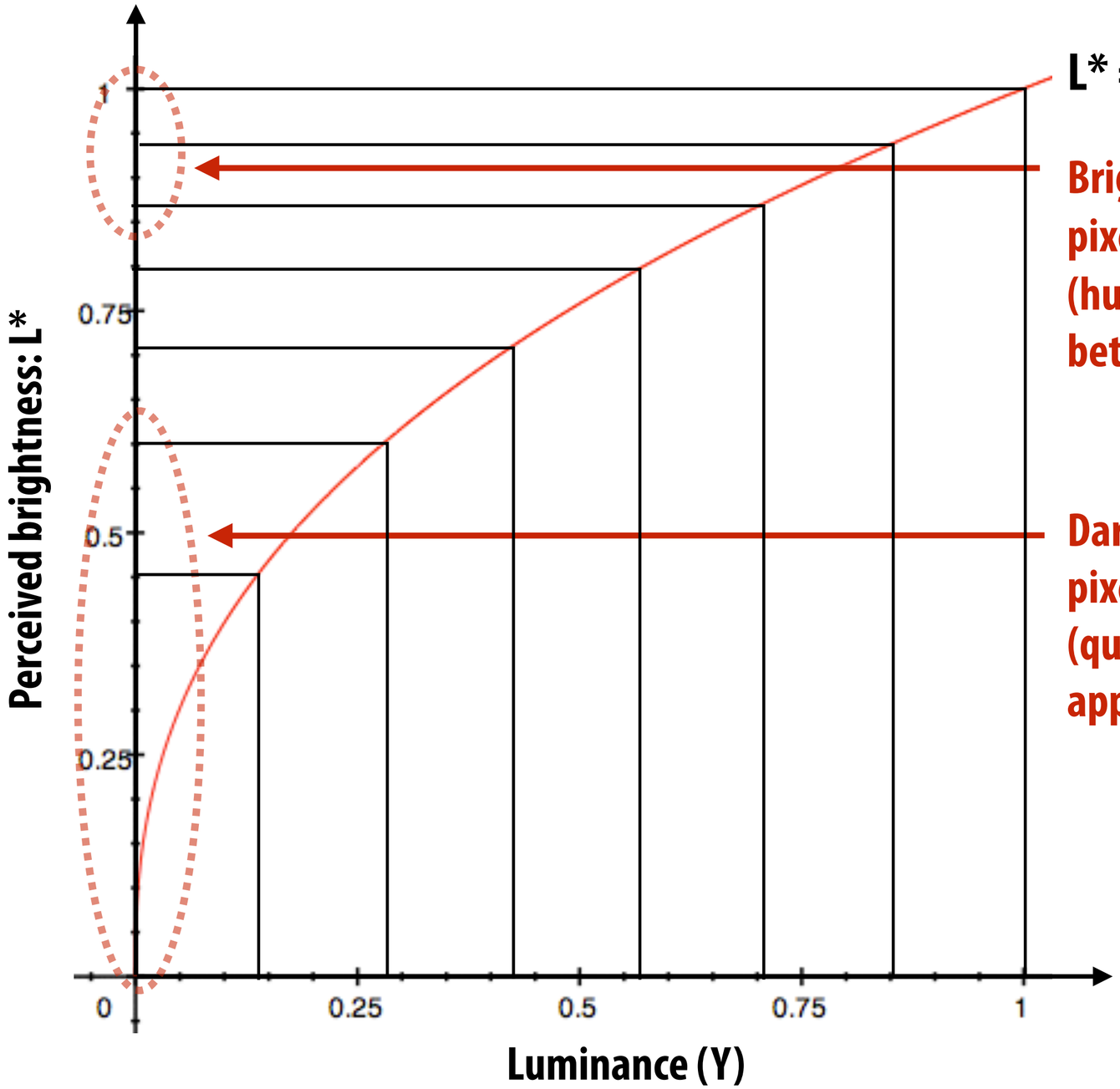
Consider 12-bit sensor pixel:
Can represent 4096 unique luminance values in output image

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



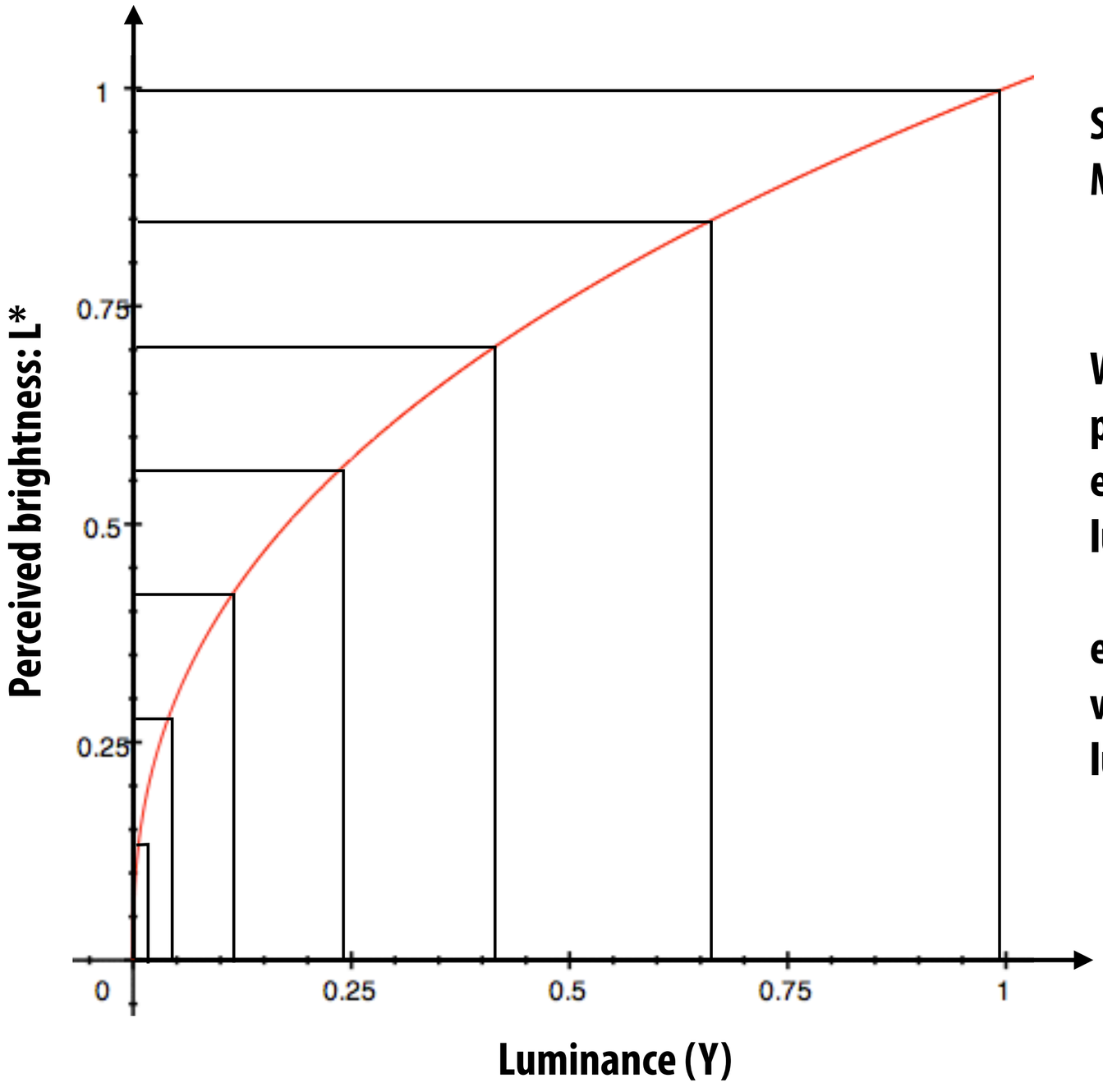
Bright regions of image: perceived difference between pixels that differ by one step in luminance is small! (human may not even be able to perceive difference between pixels that differ by one step in luminance!)

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 $(pixel\_value)^{2.2}$ 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

- Gamma Correction (Non-linear mapping)
- Color Space Conversion (Y'CbCr)

3x8-bits per pixel
Pixel values perceptually linear

- Chroma Subsampling
- JPEG compression

Takeways

- The values of pixels in photograph 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 the image signal processing ASIC in most cameras today

- In the coming lectures we'll discuss more advanced image processing operations that are emerging in modern camera pipelines
  - Local contrast enhancement, advanced denoising, high-dynamic range imaging, etc.
  - Growing sophistication and diversity of techniques suggests that current ISPs will likely become more programmable in the near future

Qualcomm Snapdragon 820
Image Signal Processor (ISP):
ASIC for processing pixels off camera
(25MP at 30Hz)