Lecture 3:

The Camera Image Processing Pipeline (Part II)

Visual Computing Systems

Stanford CS348K, Spring 2023

Theme of previous and this lecture…

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

Computation (computer graphics, image processing, and ML) is a fundamental aspect of producing high-quality photographs.

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

Beautiful image that impresses your Instagram friends

Picking up from last time…

Demosiac

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

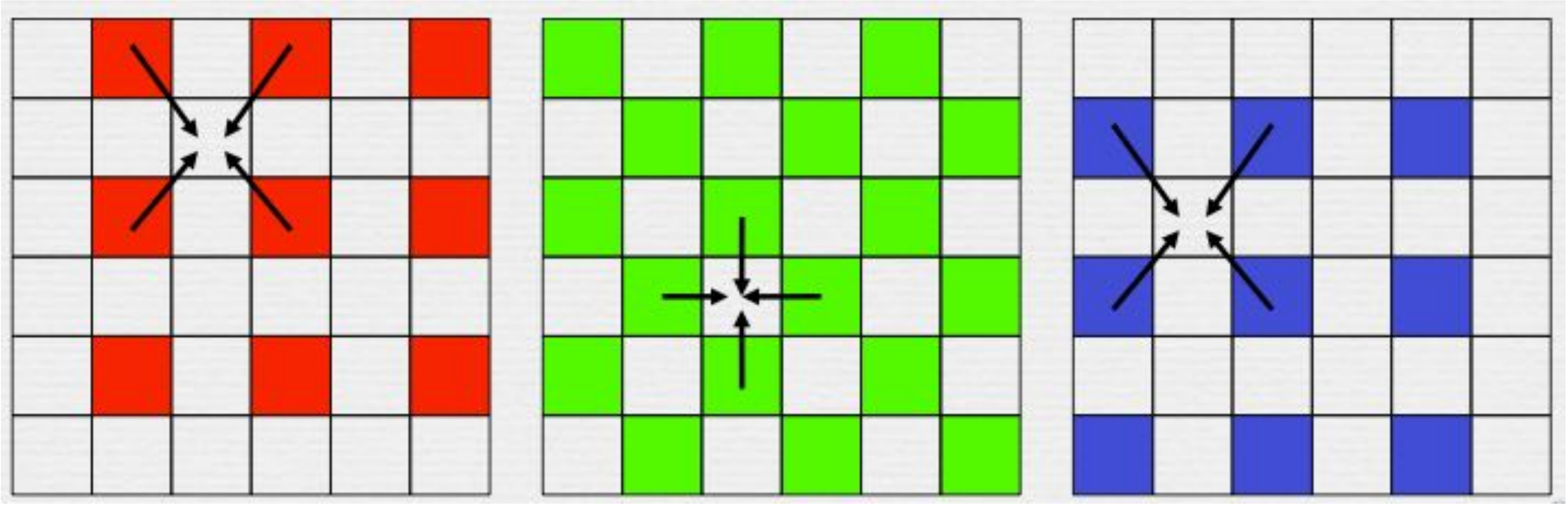
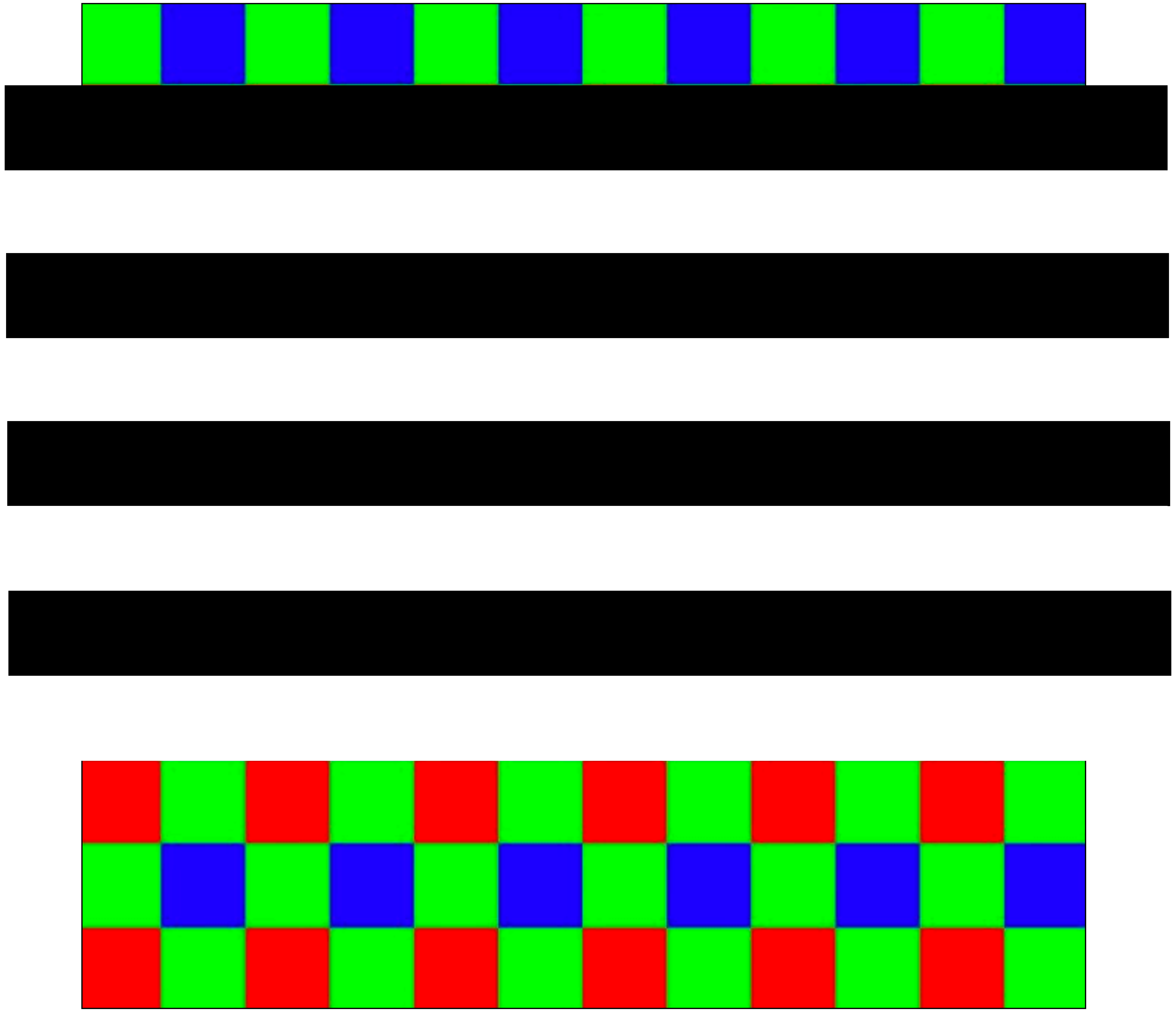


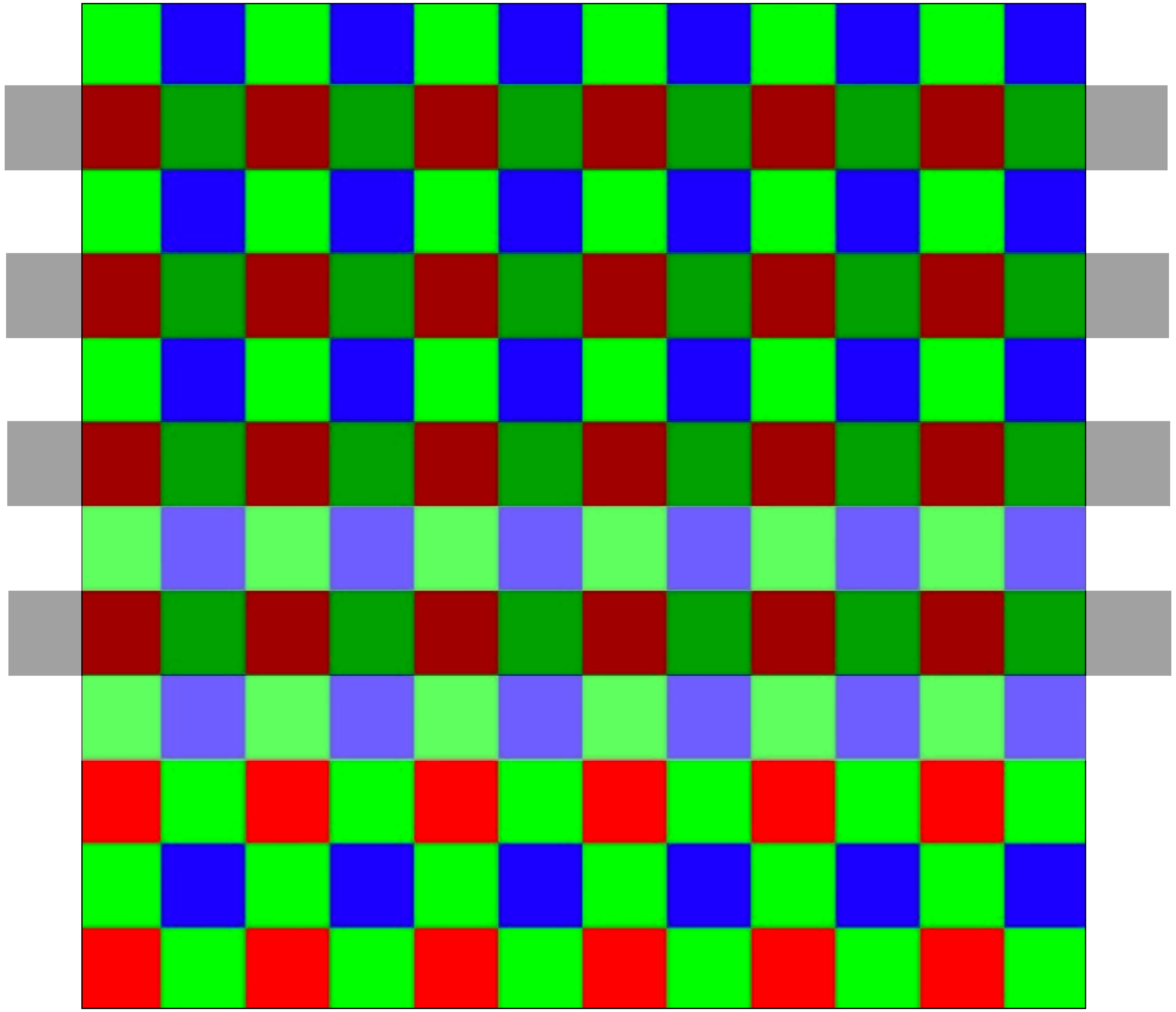
Image credit: Mark Levoy

Demosaicing errors



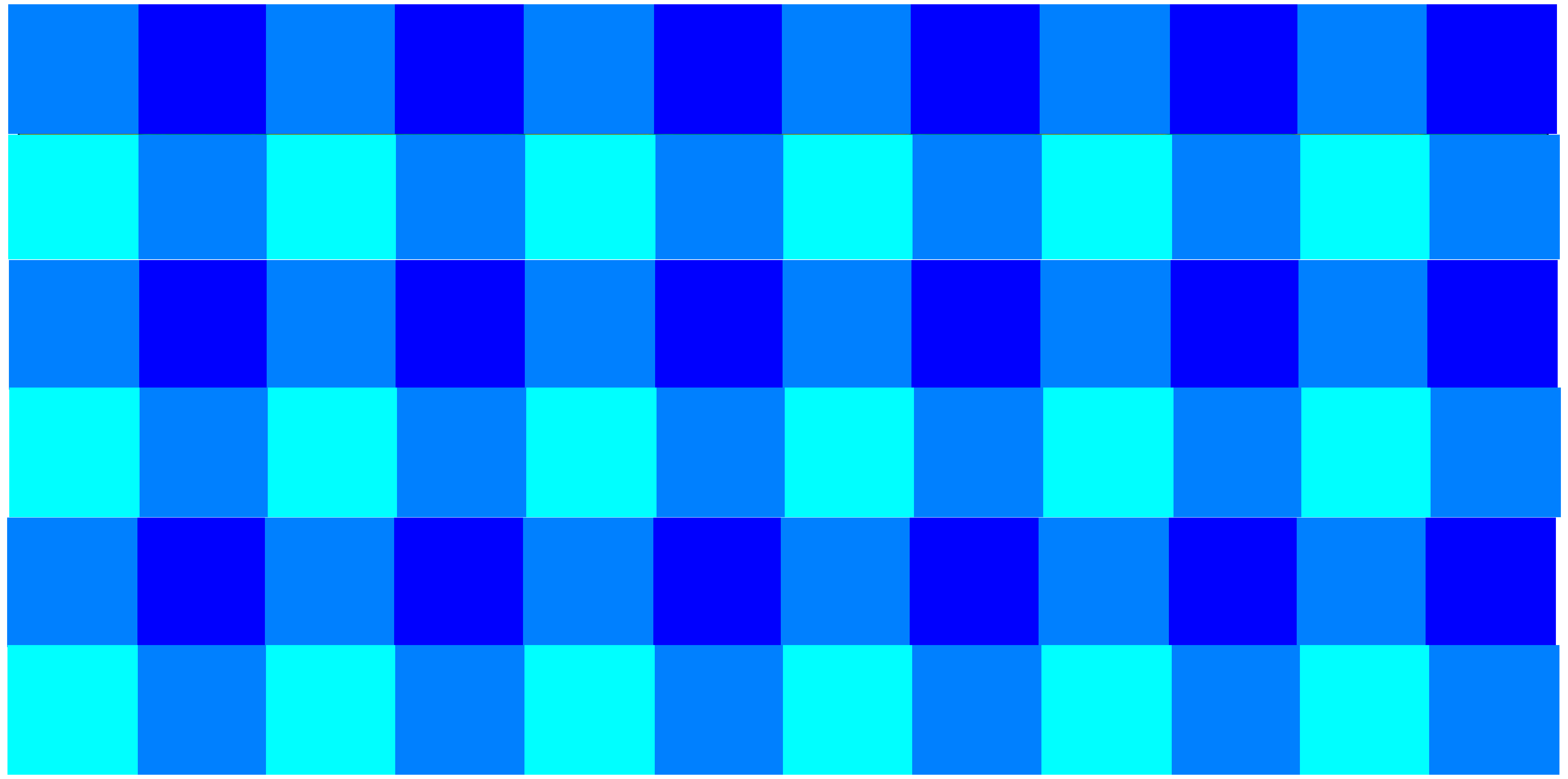
What will demosaiced result look like if this black and white signal was captured by the sensor?

Demosaicing errors



(Visualization of signal and Bayer pattern)

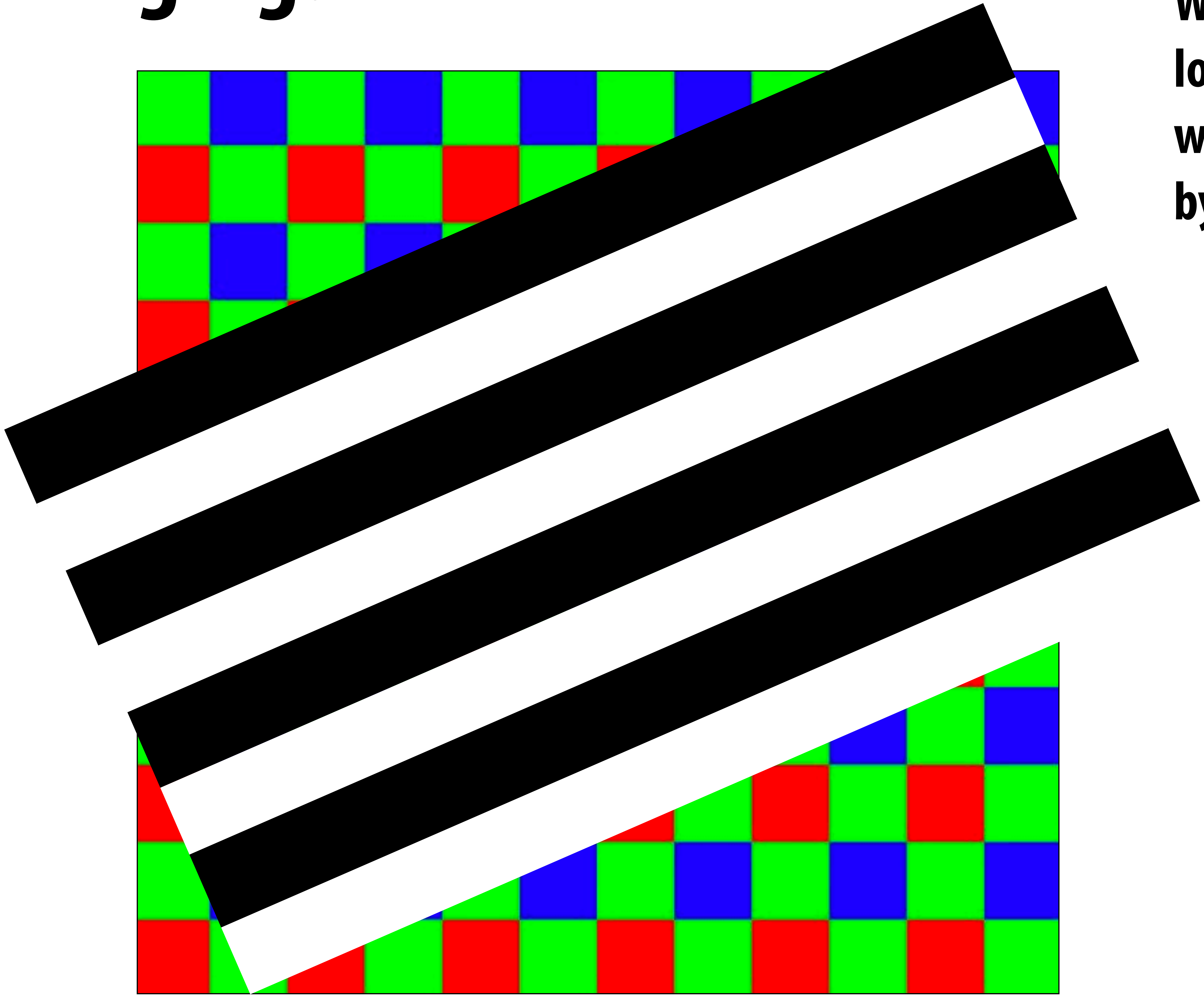
Demosaicing errors



No red measured.

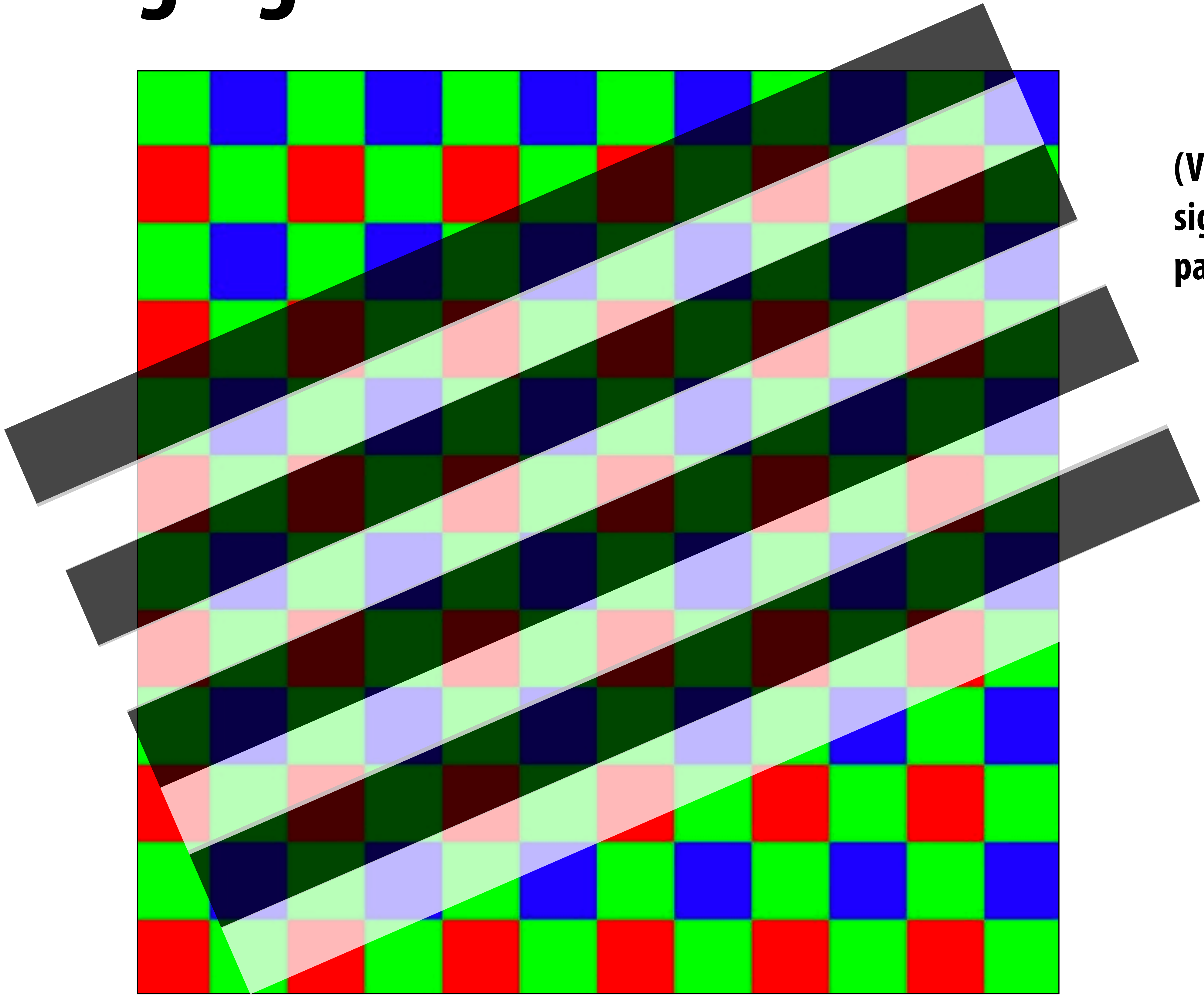
Interpolation of green yields dark/light pattern.

Why color fringing?



What will demosaiced result look like if this black and white signal was captured by the sensor?

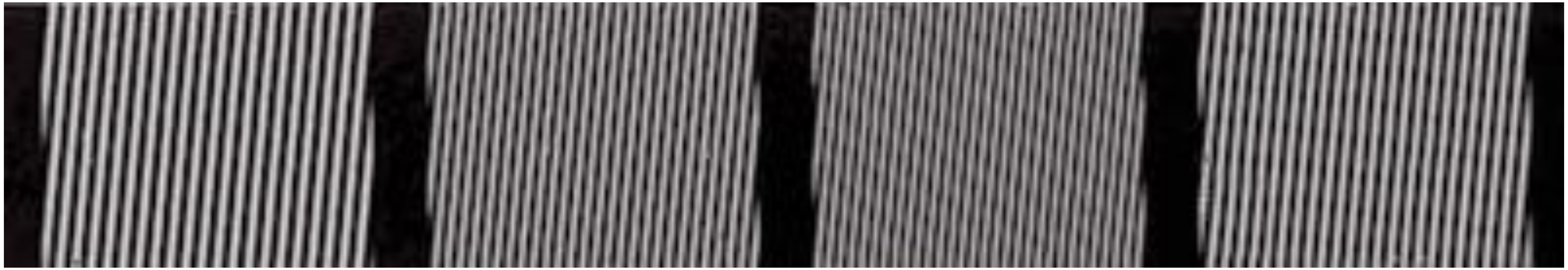
Why color fringing?



(Visualization of signal and Bayer pattern)

Demosaicing errors

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



RAW data
from sensor

RGB result after
demosaic

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

Y'CbCr color space

Colors are represented as point in 3-space

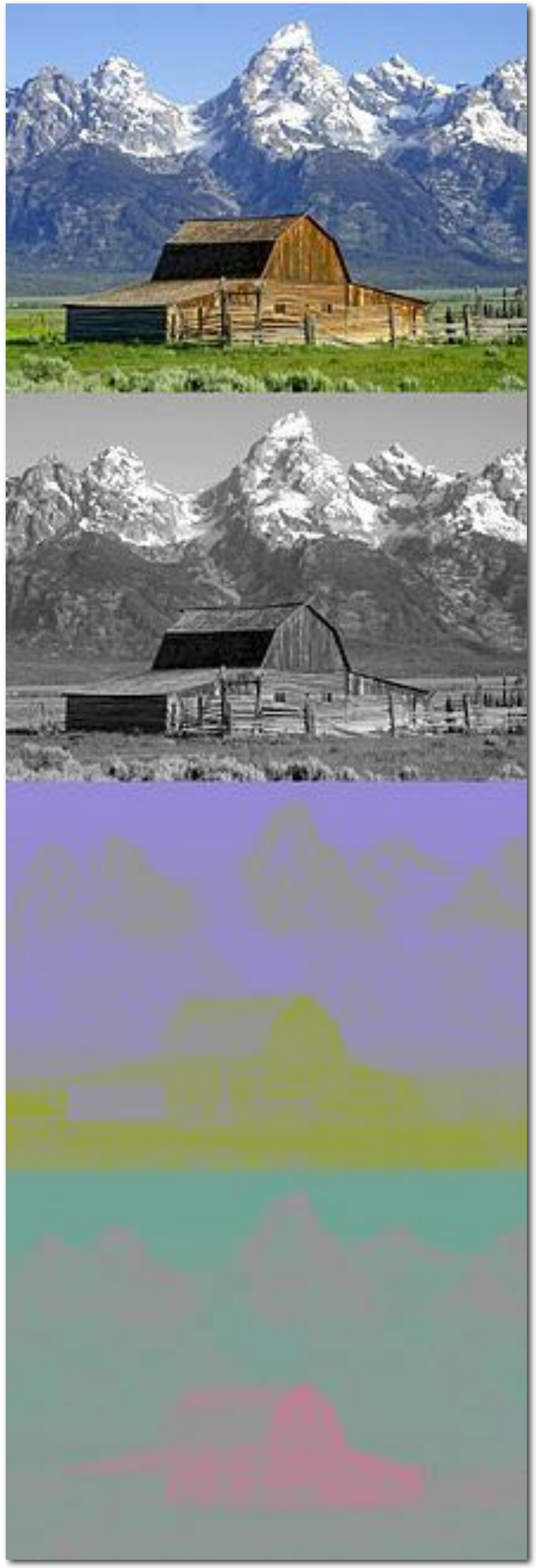
RGB is just one possible basis for representing color

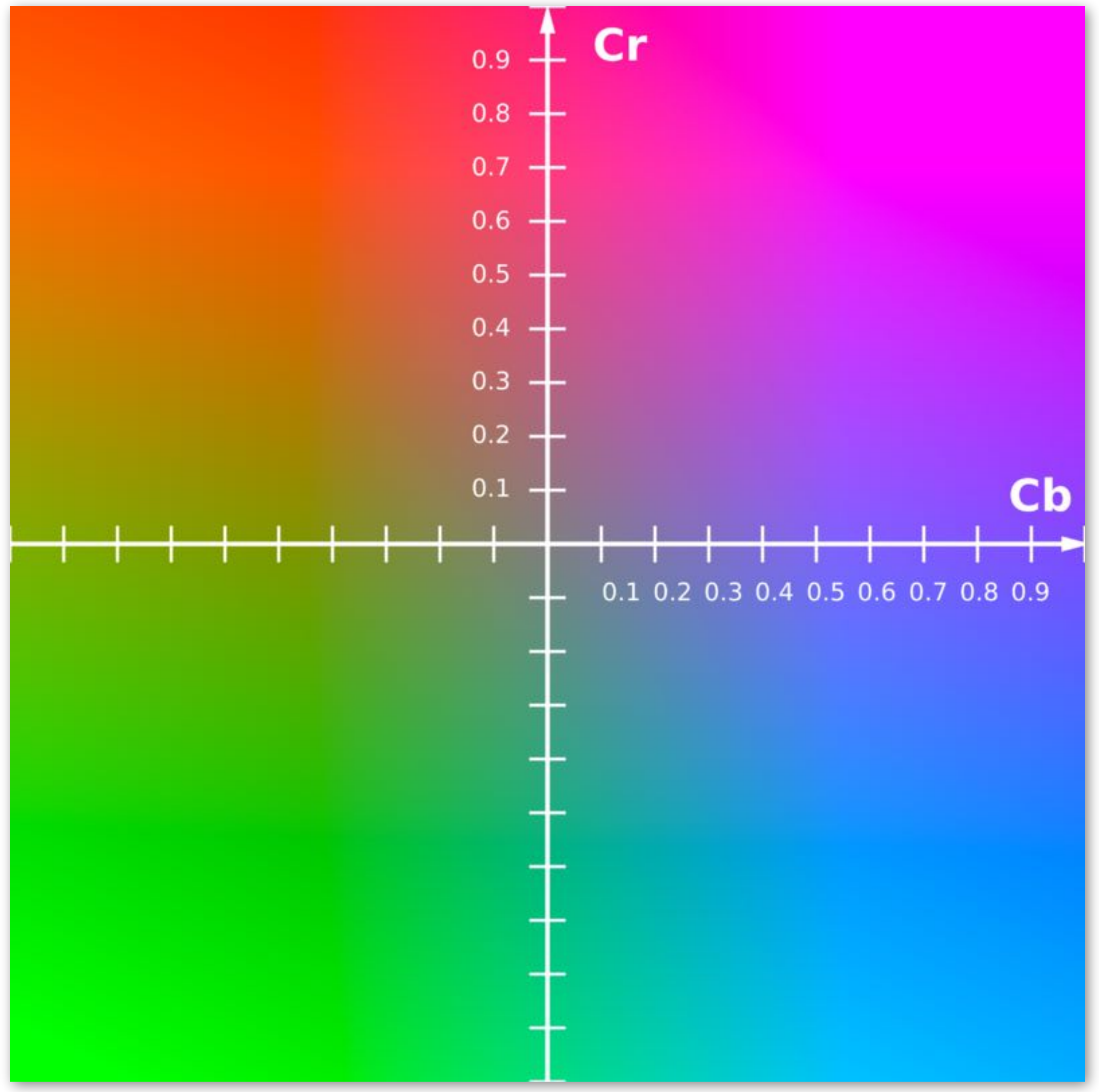
Y'CbCr separates luminance from hue in representation

Y'

Cb

Cr





Y' = luma: perceived luminance

Cb = blue-yellow deviation from gray

Cr = red-cyan deviation from gray

"Gamma corrected" RGB

(primed notation indicates perceptual (non-linear) space)

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

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

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

Image credit: Wikipedia

Example: compression in Y'CbCr

Original picture of Kayvon

Example: compression in Y’CbCr

Contents of CbCr color channels downsampled by a factor of 20 in each dimension
(400x reduction in number of samples)

Example: compression in Y'CbCr

Full resolution sampling of luma (Y')

Example: compression in Y'CbCr

Reconstructed result
(looks pretty good)

Better demosaic

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

- Trades off spatial resolution of chroma information to avoid objectionable color fringing

White balance

- Adjust relative intensity of rgb values (goal: make neutral tones in scene appear neutral in image)

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

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

Image credit: basedigitalphotography.com

White balance example

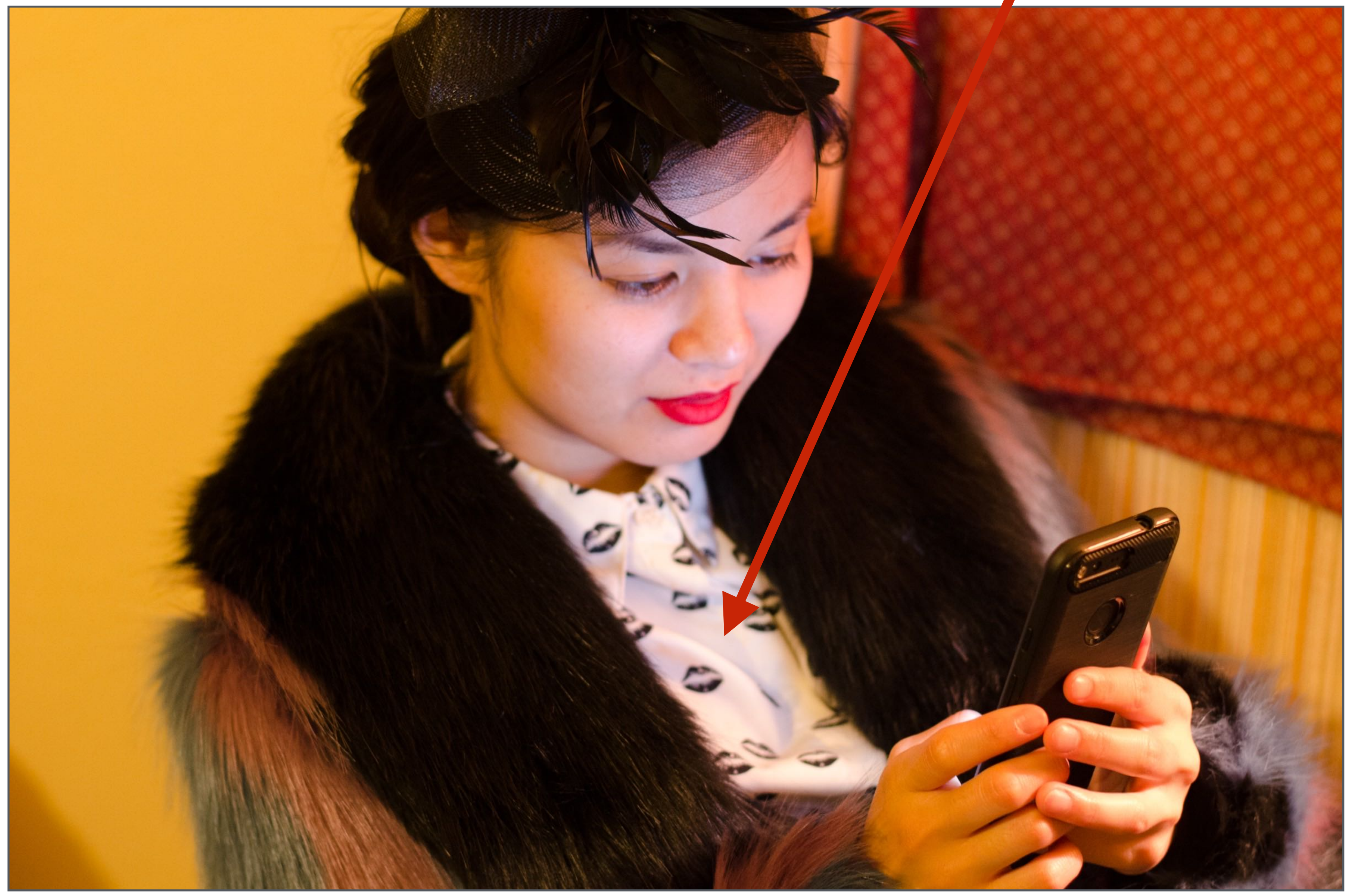
White balance example

White balance example

White balance algorithms

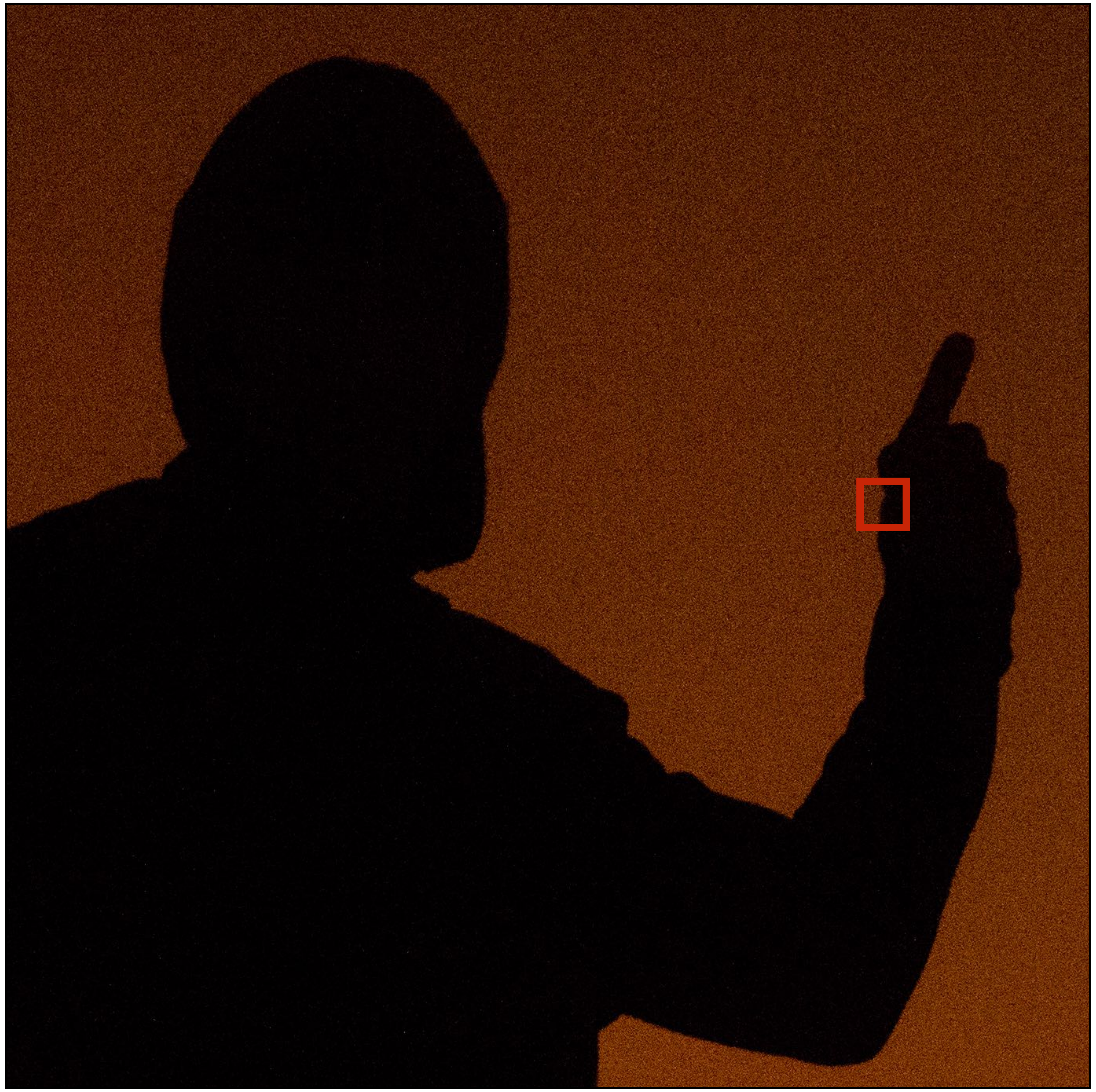
- White balance coefficients depend on analysis of image contents
  - Calibration based: get value of pixel of "white" object: (r_w, g_w, b_w)
    - Scale all pixels by $(1/r_w, 1/g_w, 1/b_w)$
  - Heuristic based: camera must guesse which pixels correspond to white objects in scene
    - Gray world assumption: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

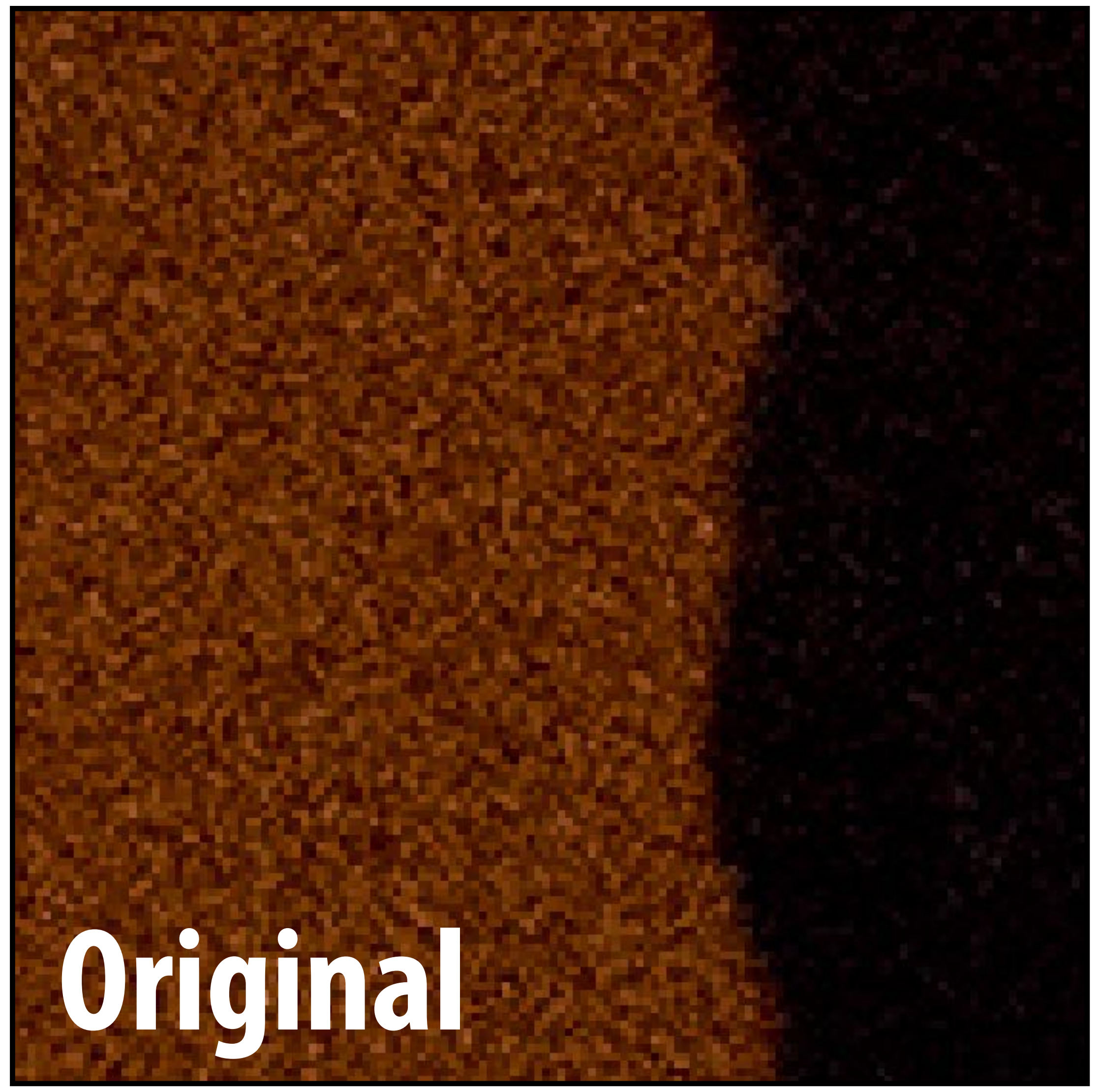
Scale r,g,b values so these pixels are close to (1,1,1)



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

Denoising






Low light conditions need long exposure…
blur due to camera shake
Image credit: https://www.colorexpertsbd.com/blog/how-to-fix-blurry-photos-induced-by-camera-shake-in-photoshop

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

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

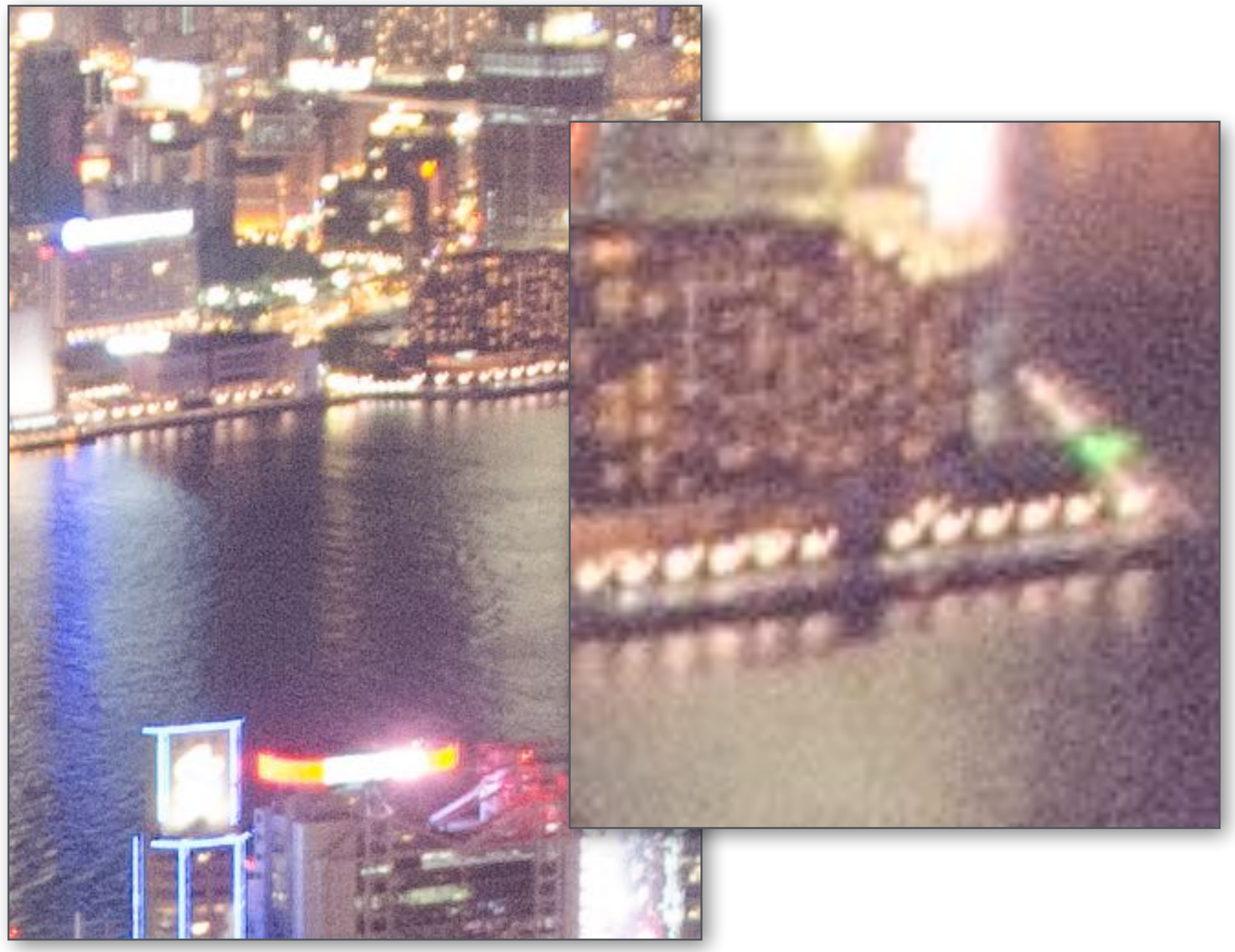
Attempt to denoise… splotchy effect remains

Long exposure: walking people are blurred…

Long exposure: walking people are blurred…

Also: still significant noise in dark regions

Reduce noise via image processing: denoising via downsampling



Downsample via point sampling

(noise remains)

Downsample via averaging

Noise reduced

Like a smaller number of bigger pixels!

Discrete 2D convolution

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

output image
(the result of convolving f with input image I)

filter

input image

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

Then:

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

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

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

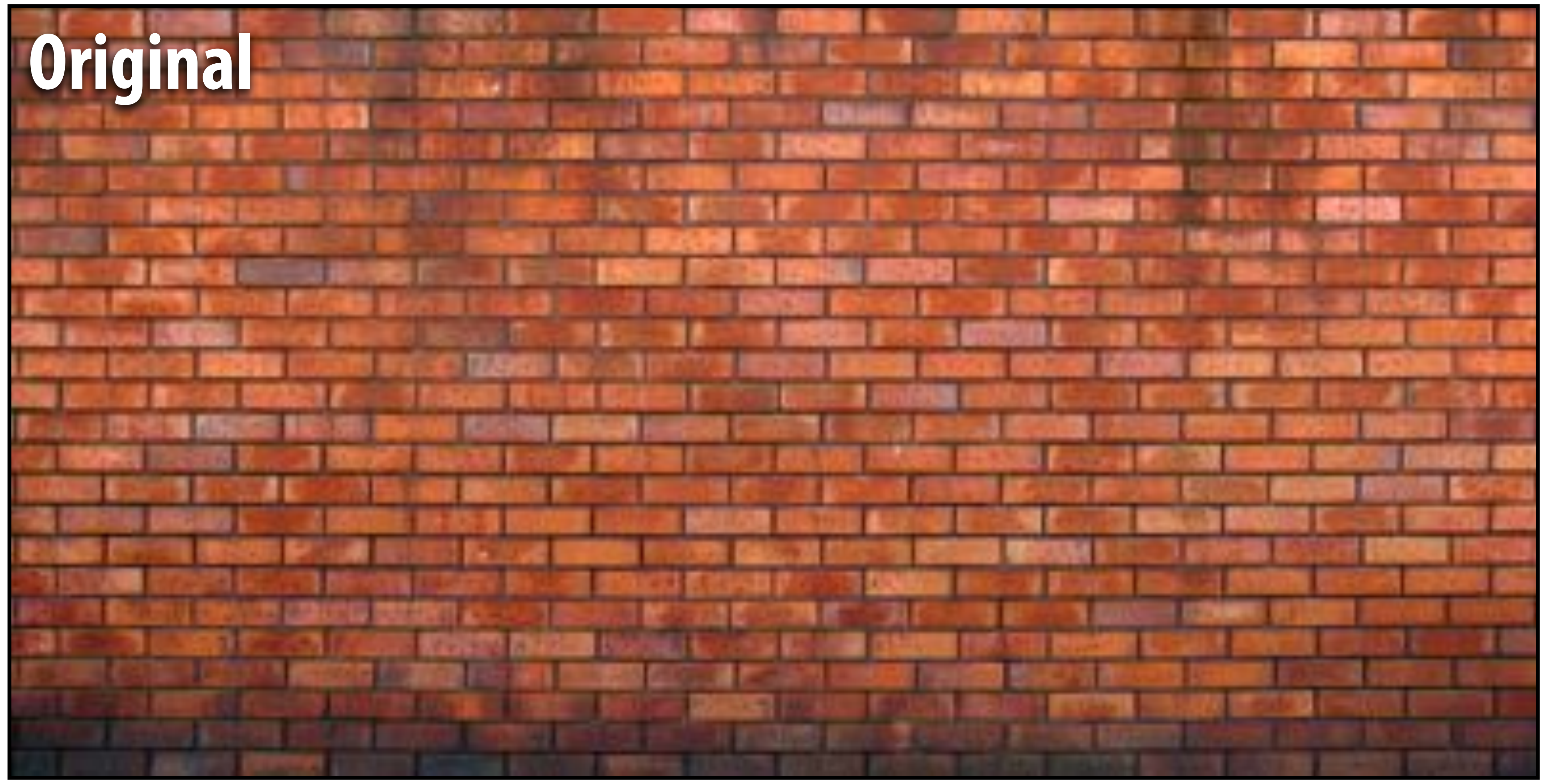
(often called: “filter weights”, “filter kernel”)

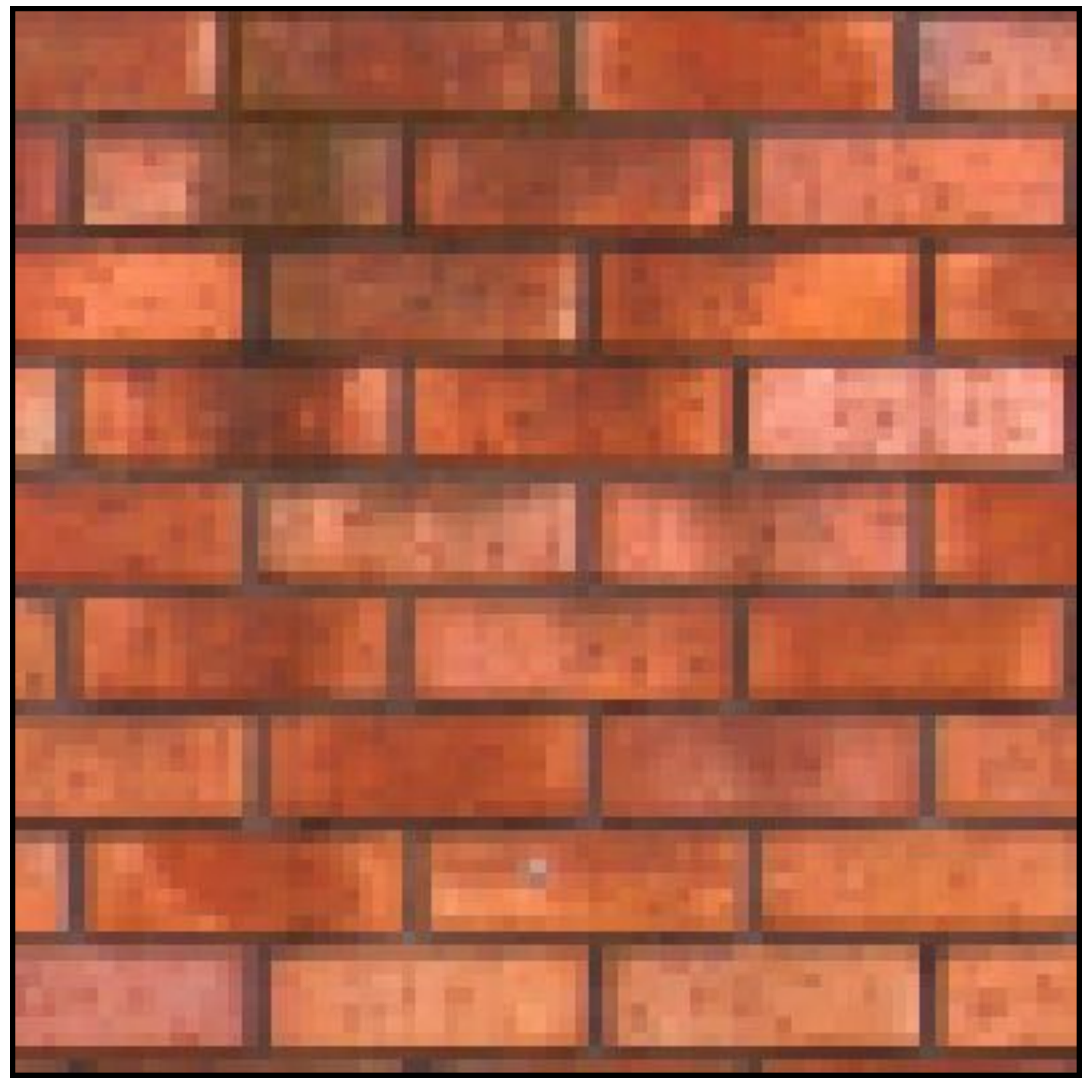
Simple 3x3 box blur in C code

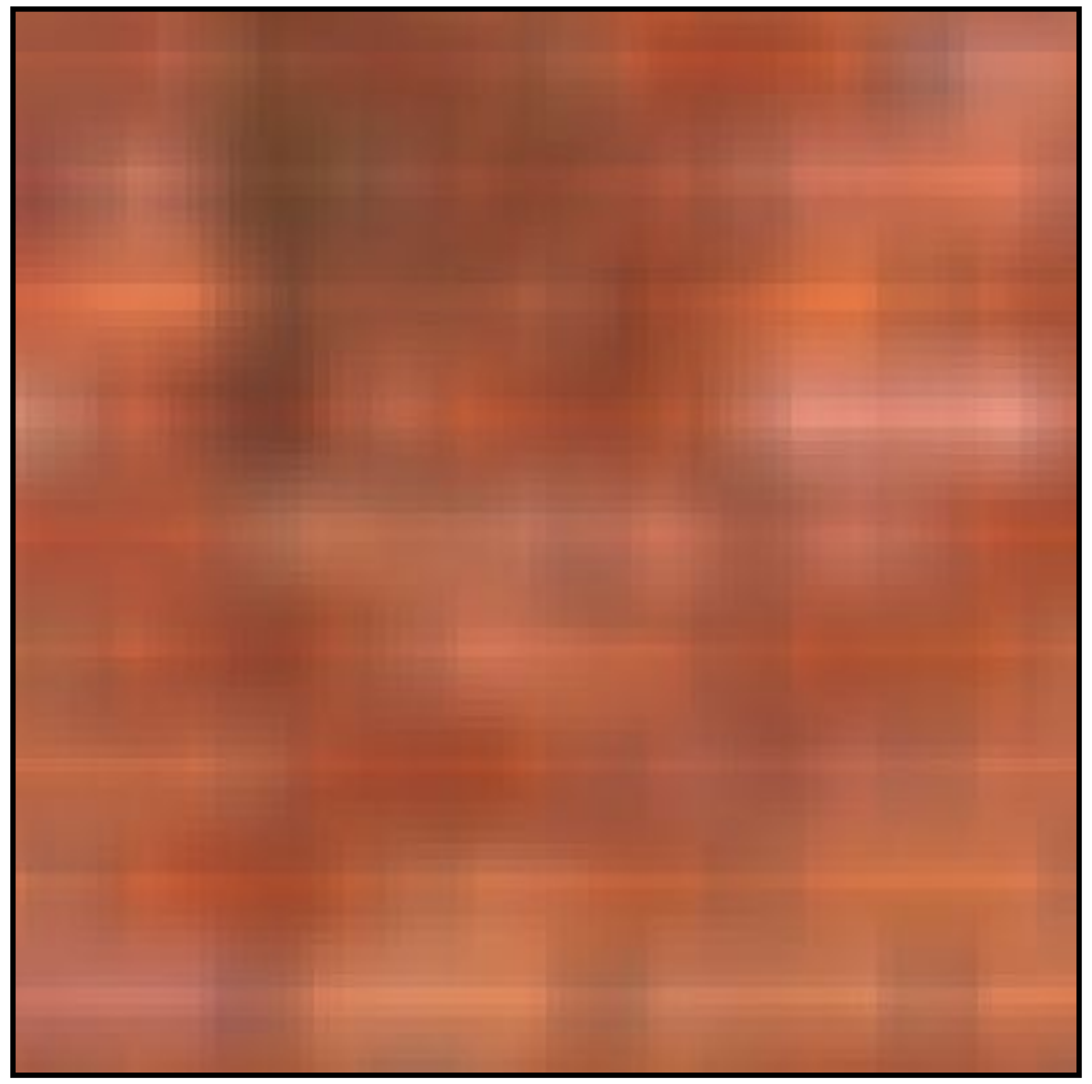
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

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


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

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

7x7 box blur







Gaussian blur

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

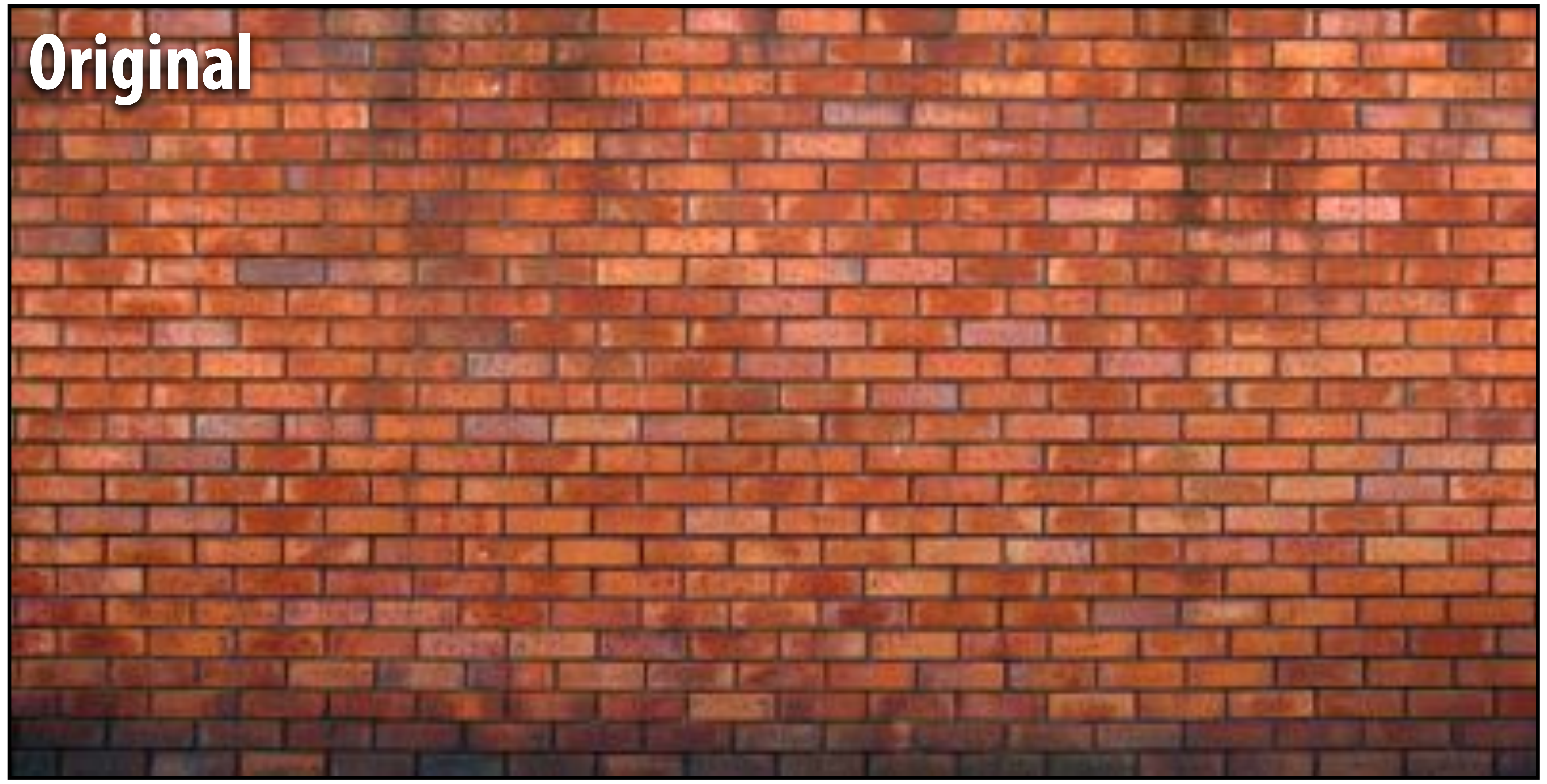
$$f(i,j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

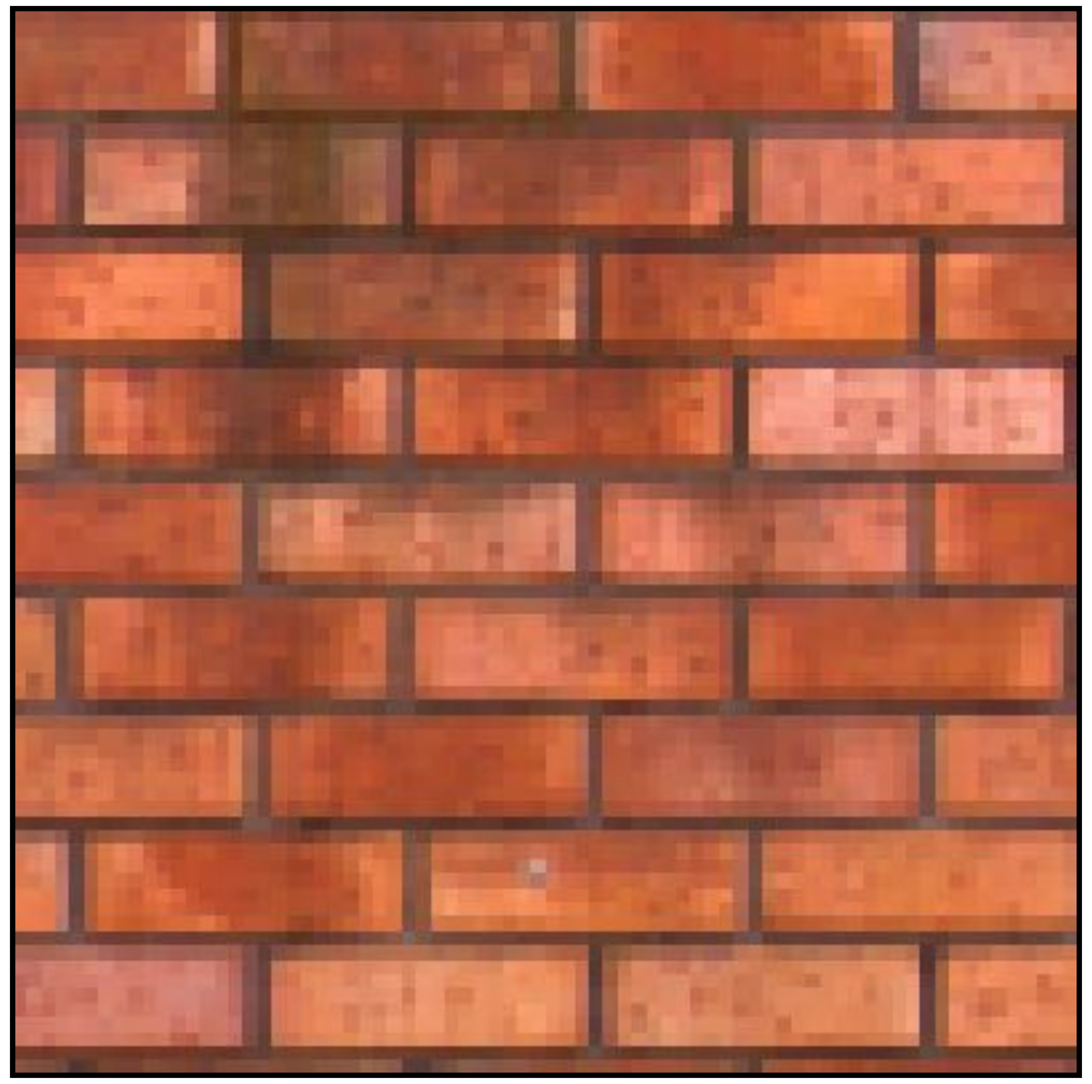
- **Produces weighted sum of neighboring pixels (contribution falls off with distance)**
  - **In practice: truncate filter beyond certain distance for efficiency**

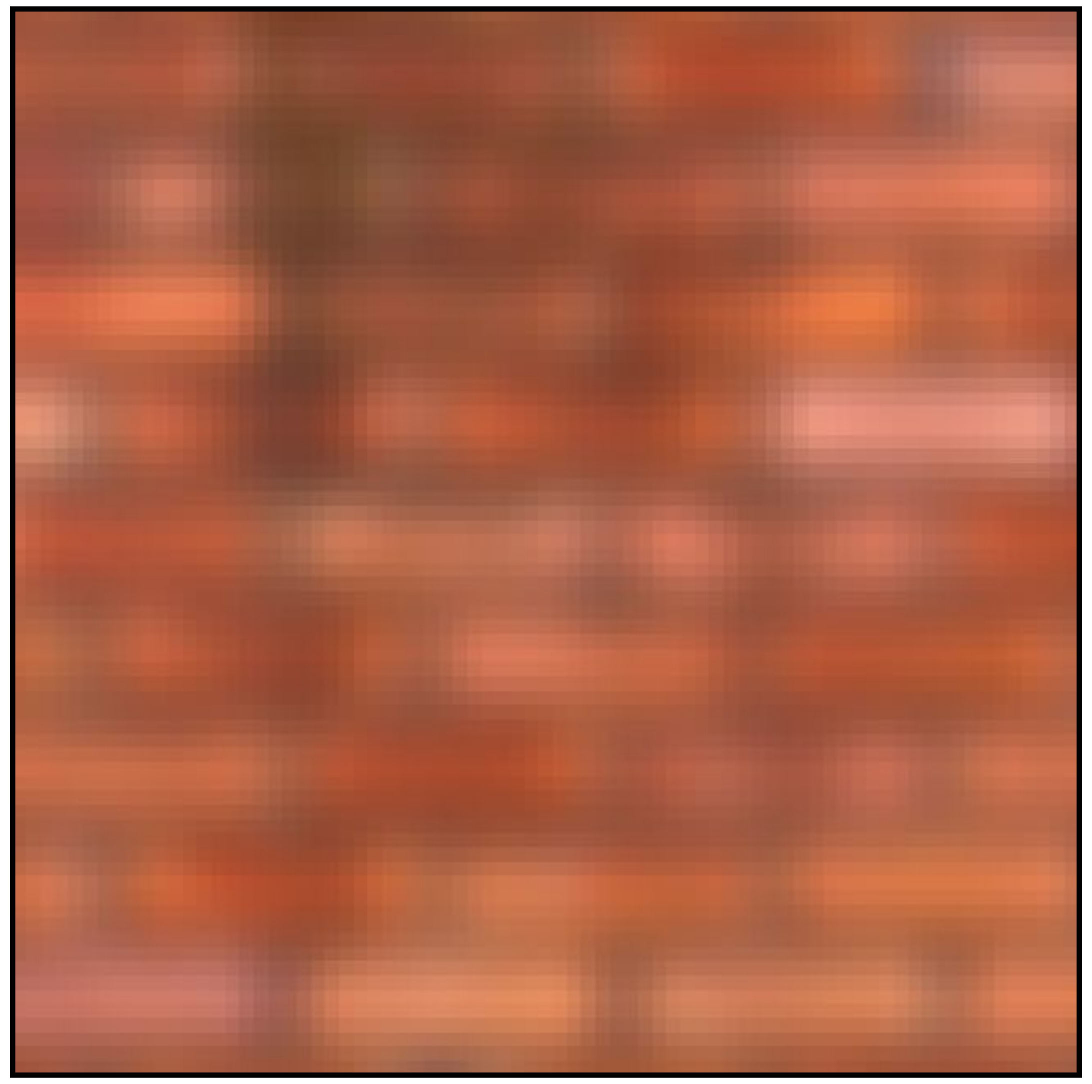
$$\frac{1}{256} \cdot \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

Note: this is a 5x5 truncated Gaussian filter

7x7 gaussian blur

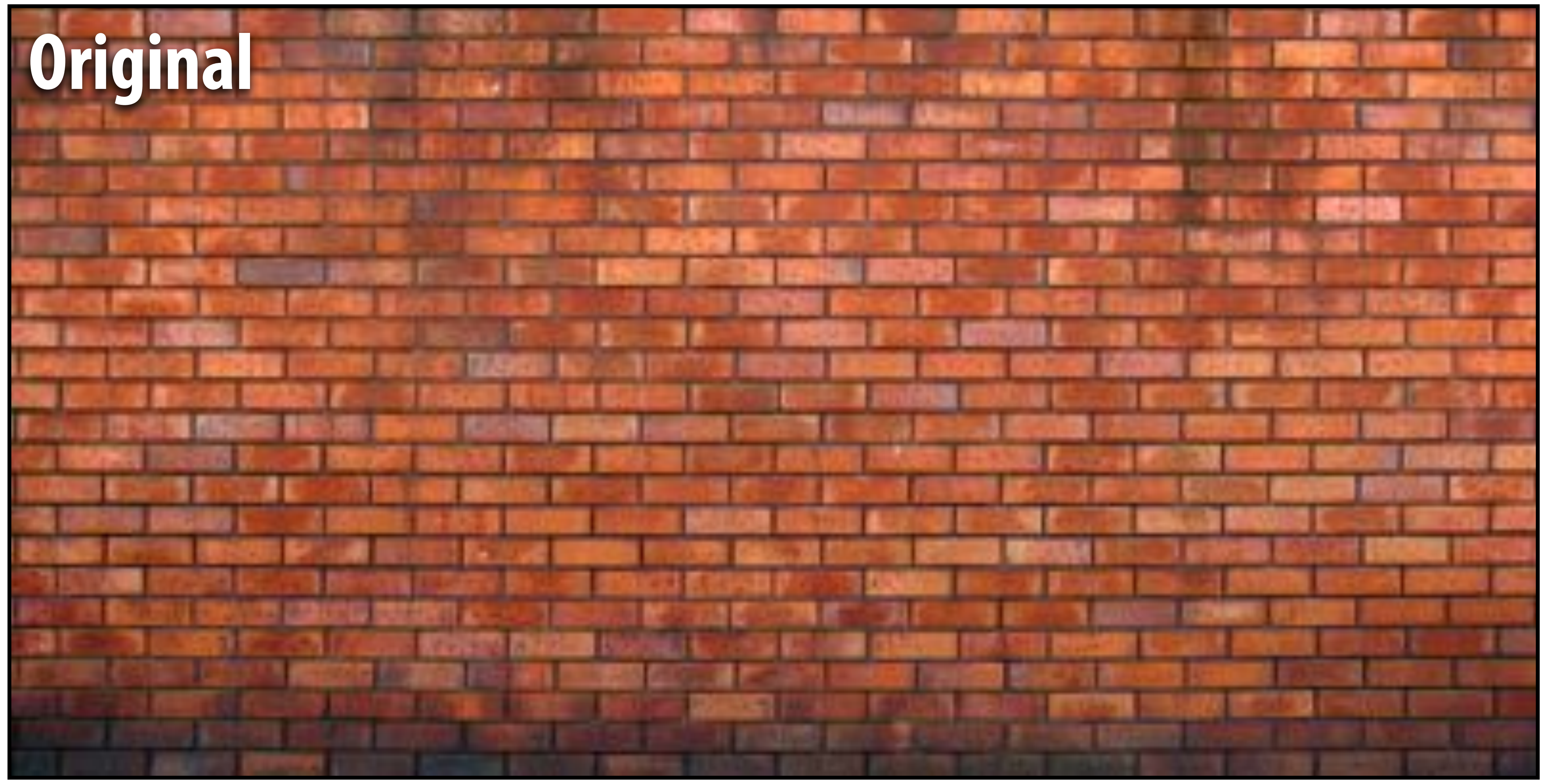


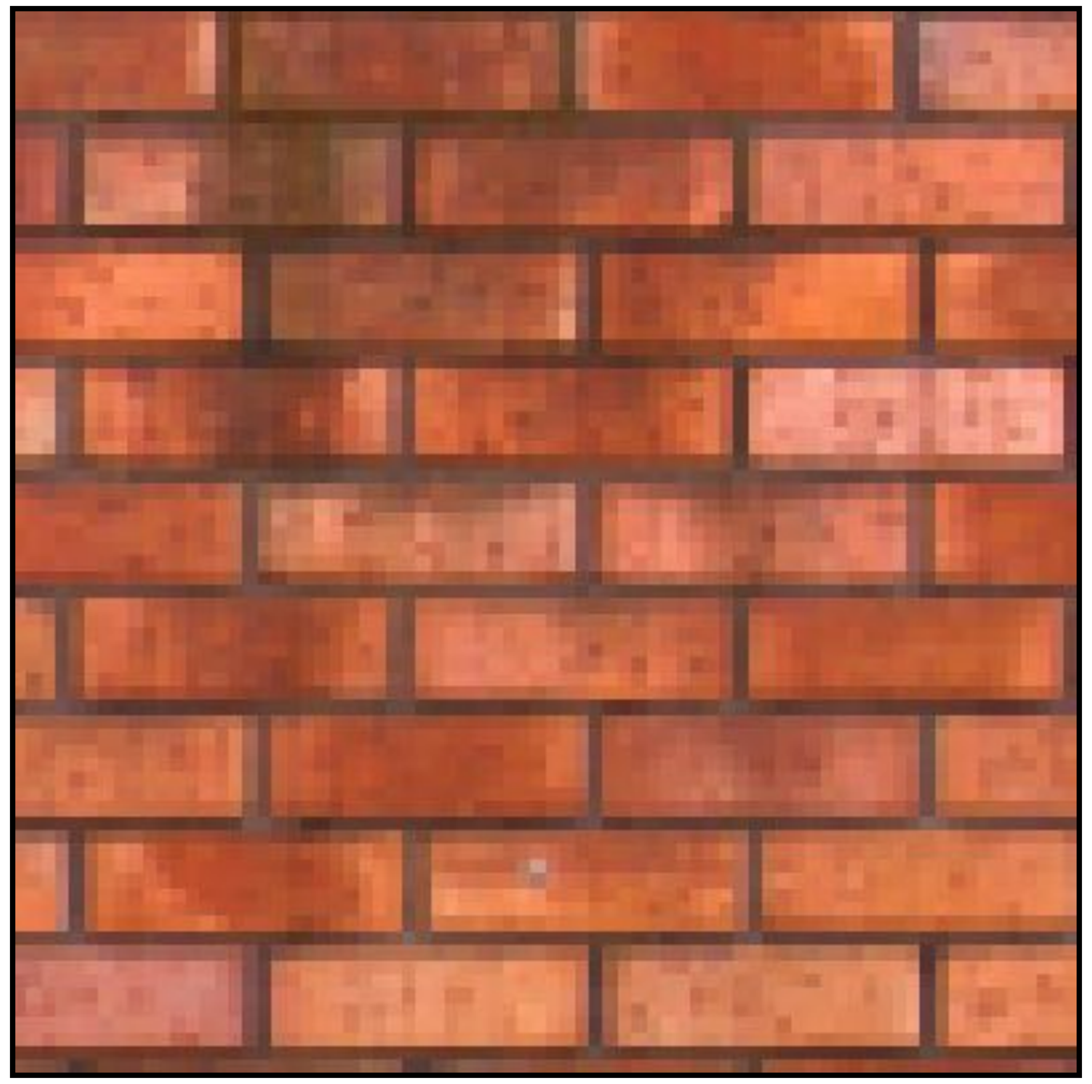


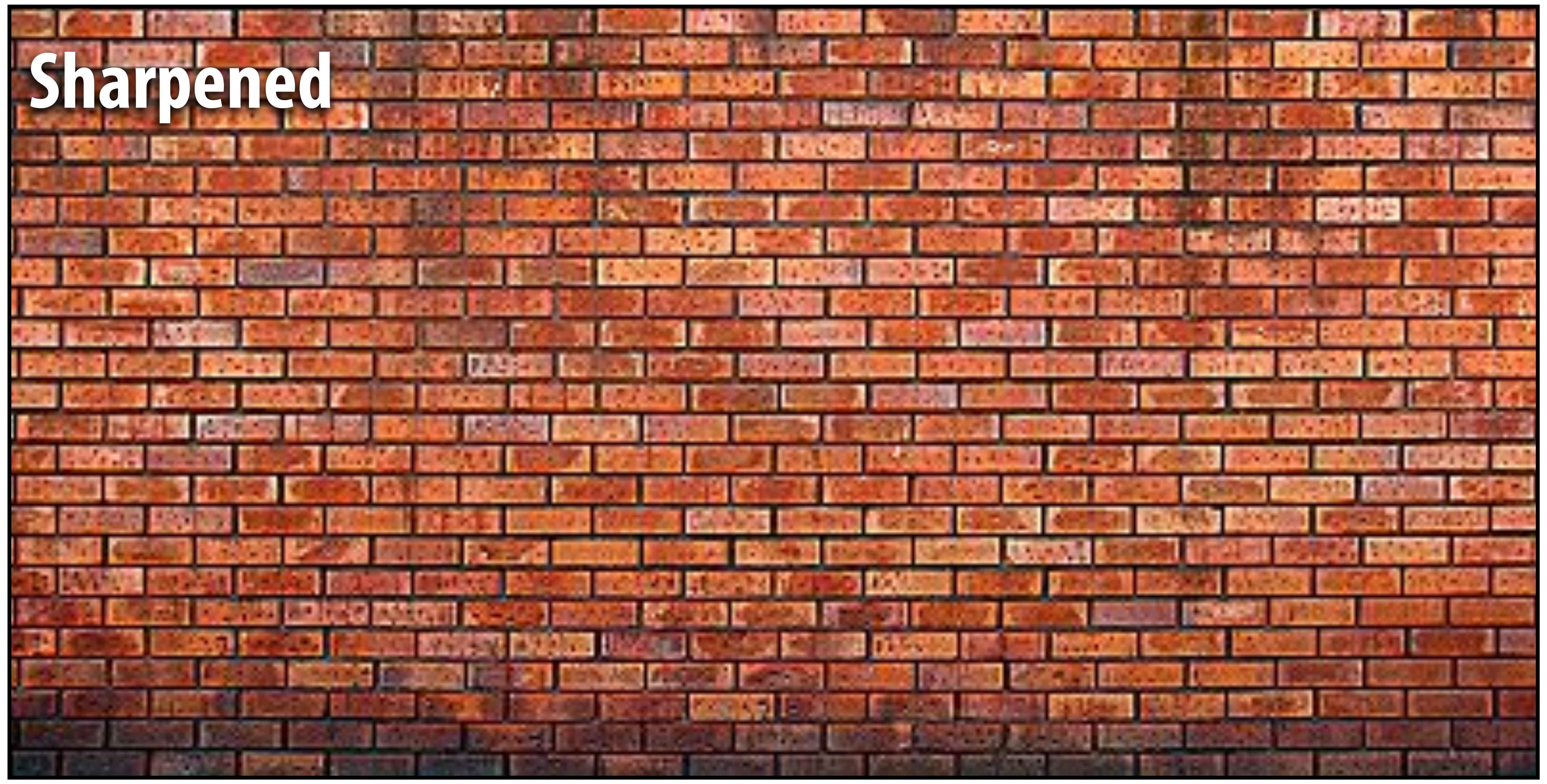


3x3 sharpen filter

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







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: filter weights are 1 or 0 (depending on image content)

original image

1px median filter

3px median filter

10px median filter

```
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, then pick median: $O(N^2 \log(N^2))$ work per pixel
  - Can you think of an $O(N^2)$ algorithm? What about $O(N)$?

Bilateral filter

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

Bilateral filter

Gaussian blur kernel

Input image

$$\mathrm{BF}[I](p) = \frac{1}{W_p} \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|) G_\sigma(i,j) I(x-i,y-j)$$

Normalization

For all pixels in support region of Gaussian kernel

Re-weight based on difference in input image pixel values

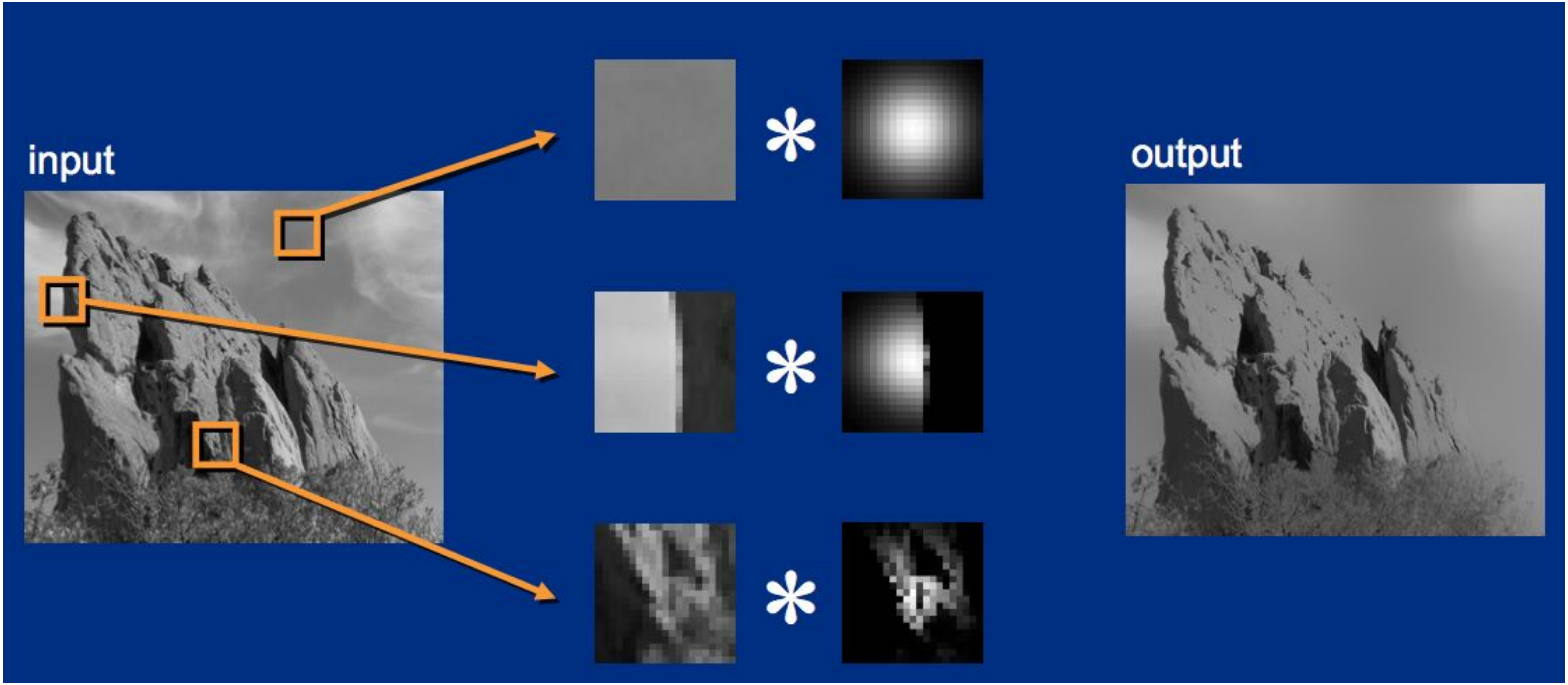
$$W_p = \sum_{i,j} f(|I(x-i,y-j) - I(x,y)|) G_\sigma(i,j)$$

- 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: $f(x) = 0$ 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 <u>spatial distance</u> and <u>input image pixel intensity</u> difference. (non-linear filter: like the median filter, the filter's weights depend on input image content)

Bilateral filter: kernel depends on image content



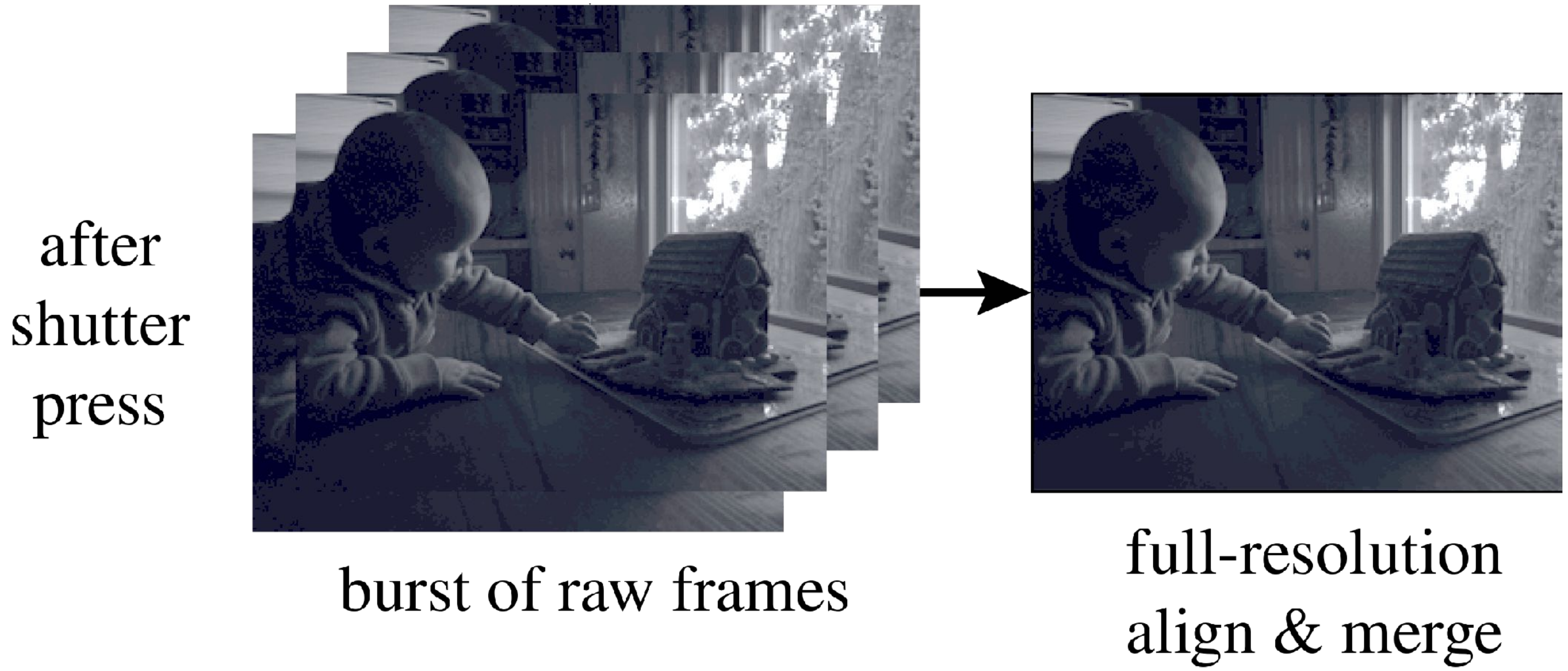
See Paris et al. [ECCV 2006] for a fast approximation to the bilateral filter

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

Better denoising idea: merge sequence of captures

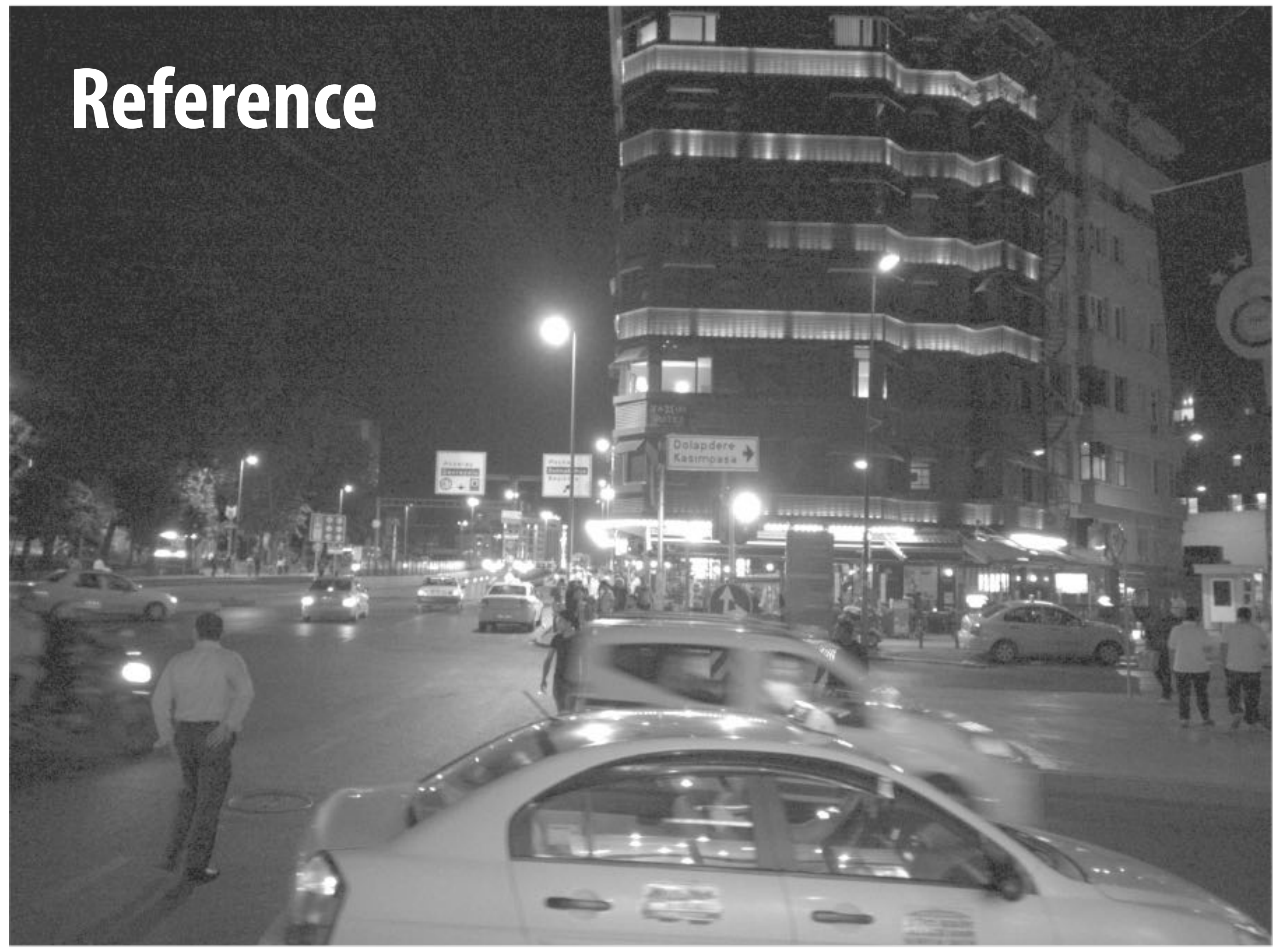
Algorithm used in Google Pixel Phones [Hasinoff 16]

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

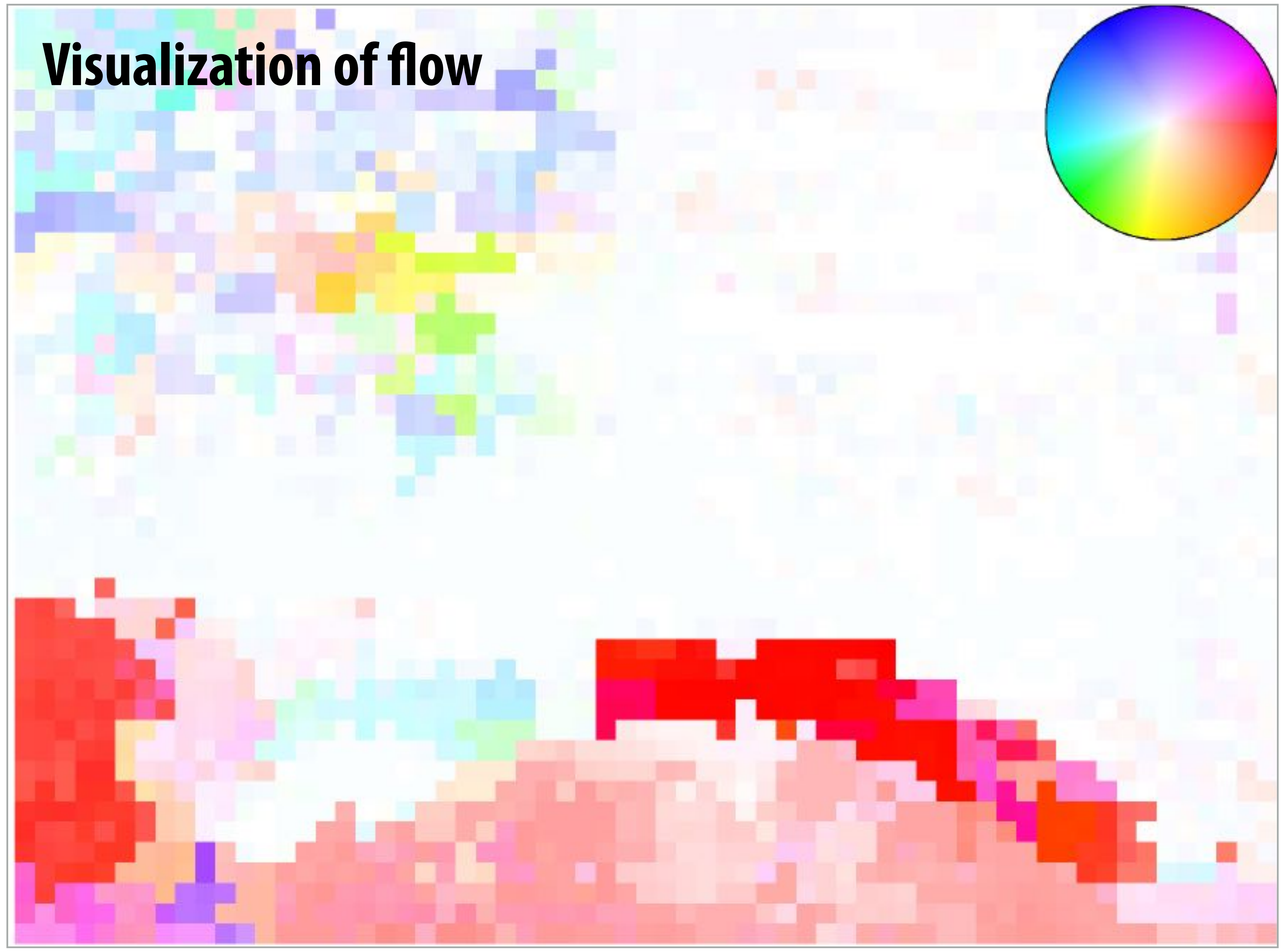


Align and merge algorithm

Image pair



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



[Image credit: Hasinoff 16]

Results of align and merge

[Hasinoff 16]

Full image

Successful alignment

Alignment failure

Reference frame | Temporal mean of images in burst (blurry) | Temporal mean with alignment | Robust merge with alignment

[Image credit: Hasinoff 16]

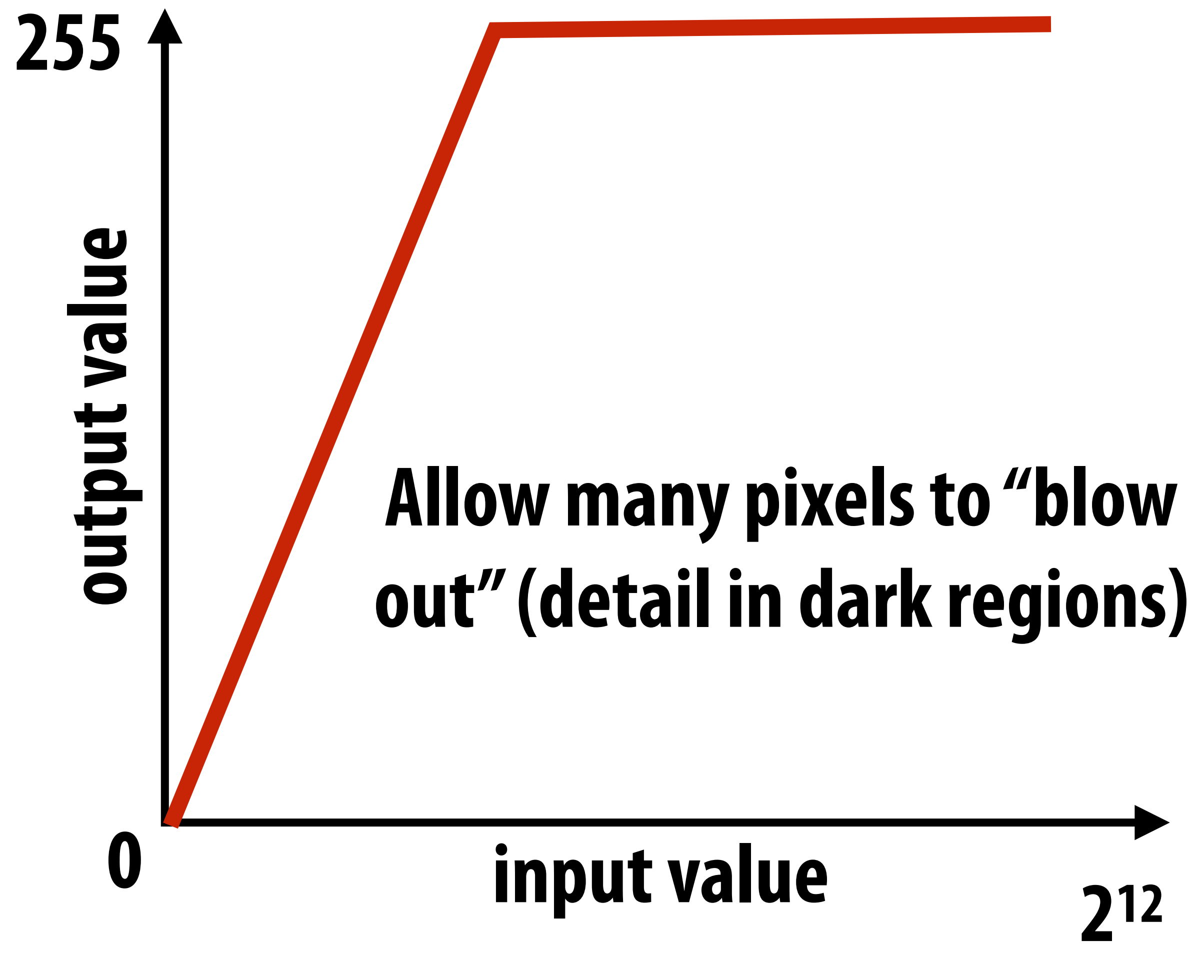
Saturated pixels

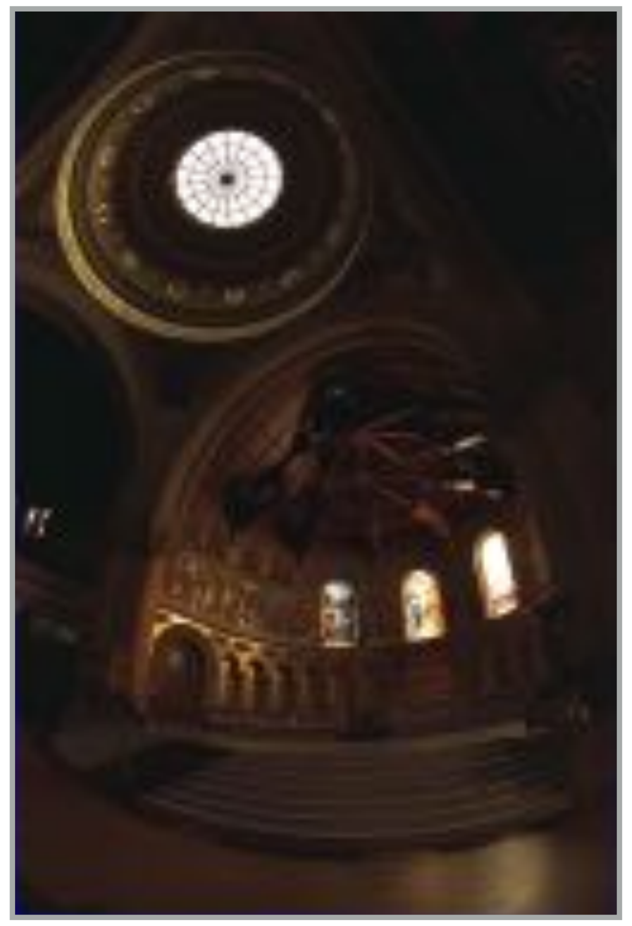
Saturated pixels

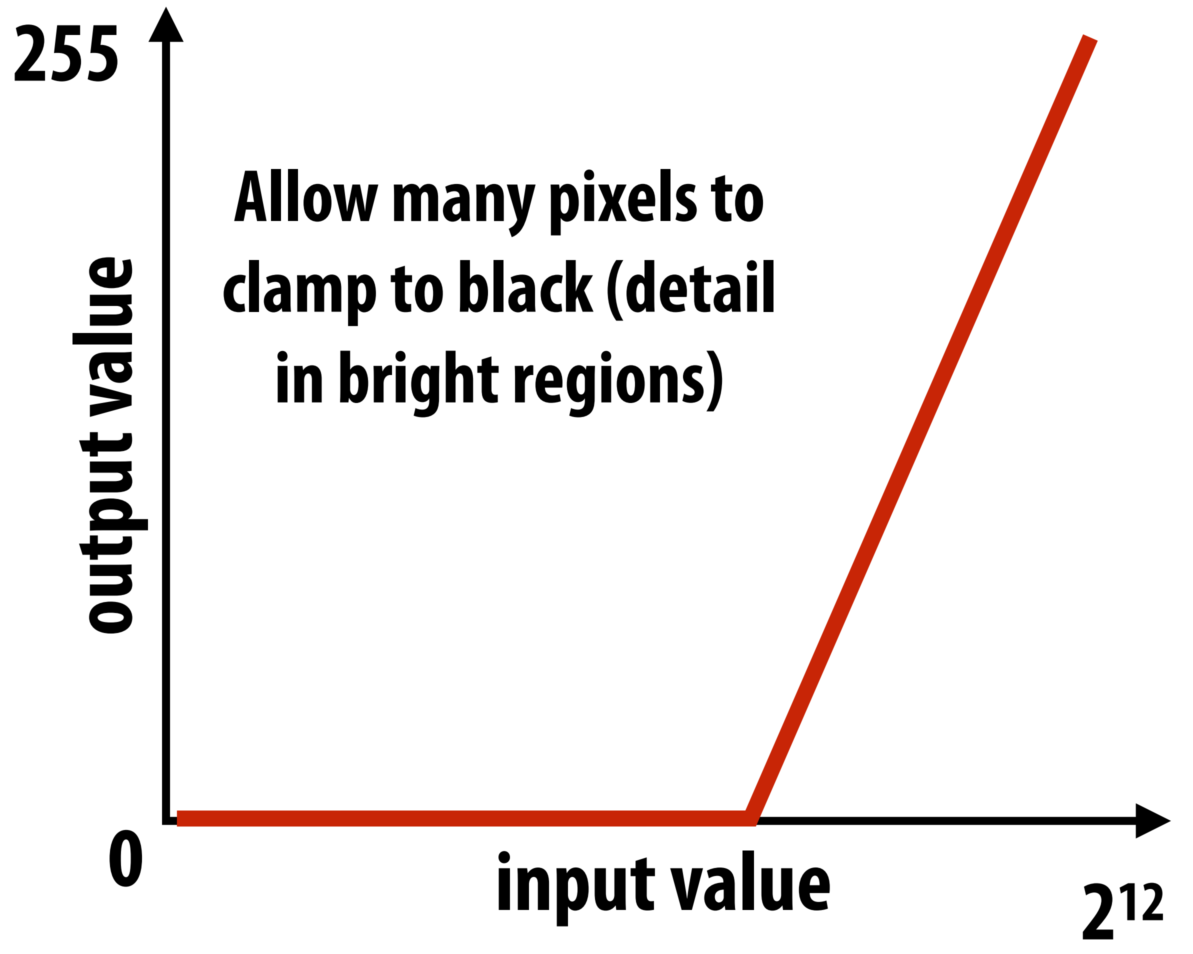
Credit: P. Debevec

Global tone mapping

- Measured image values (by camera's sensor): 10-12 bits / pixel, but common image formats are 8-bits/pixel
- How to convert 12 bit number to 8 bit number?







High dynamic range image (HDR)
Detail in dark and light images

Image credit: Wikipedia

Local tone adjustment

Pixel values

Weights

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

(no physical basis for this)

Combined image
(unique weights per pixel)

Image credit: Mertens 2007

Challenge of merging images

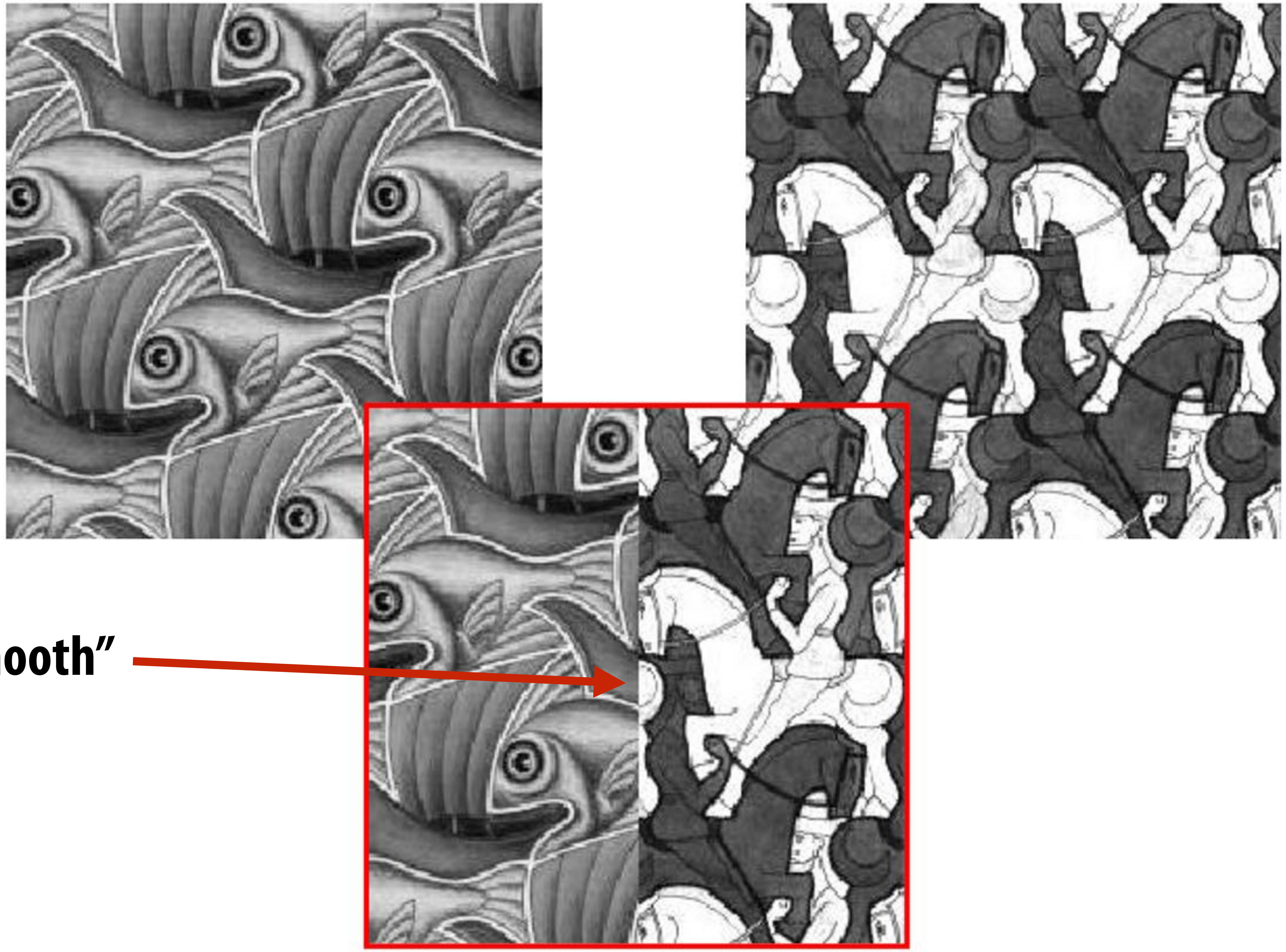
Four exposures (weights not shown)

Merged result (based on weight masks)
Notice heavy "banding" since absolute intensity
of different exposures is different

Merged result
(after blurring weight mask)
Notice "halos" near edges

Image blending

Consider a simple case where we want to blend two patterns:

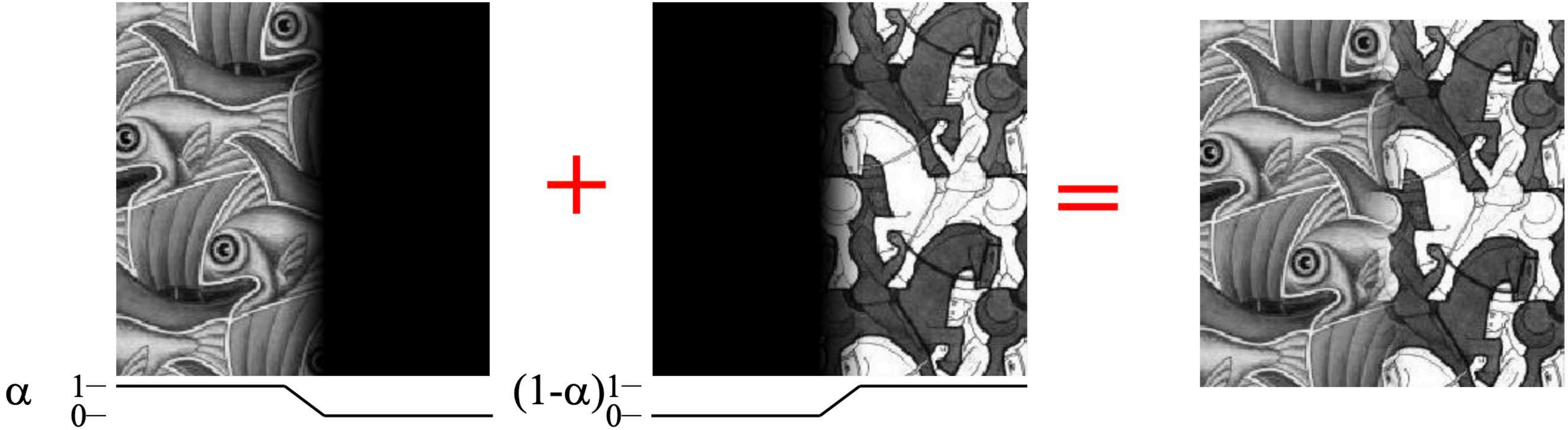


Problem: not "smooth"

Slide credit: Efros

“Feather” the alpha mask

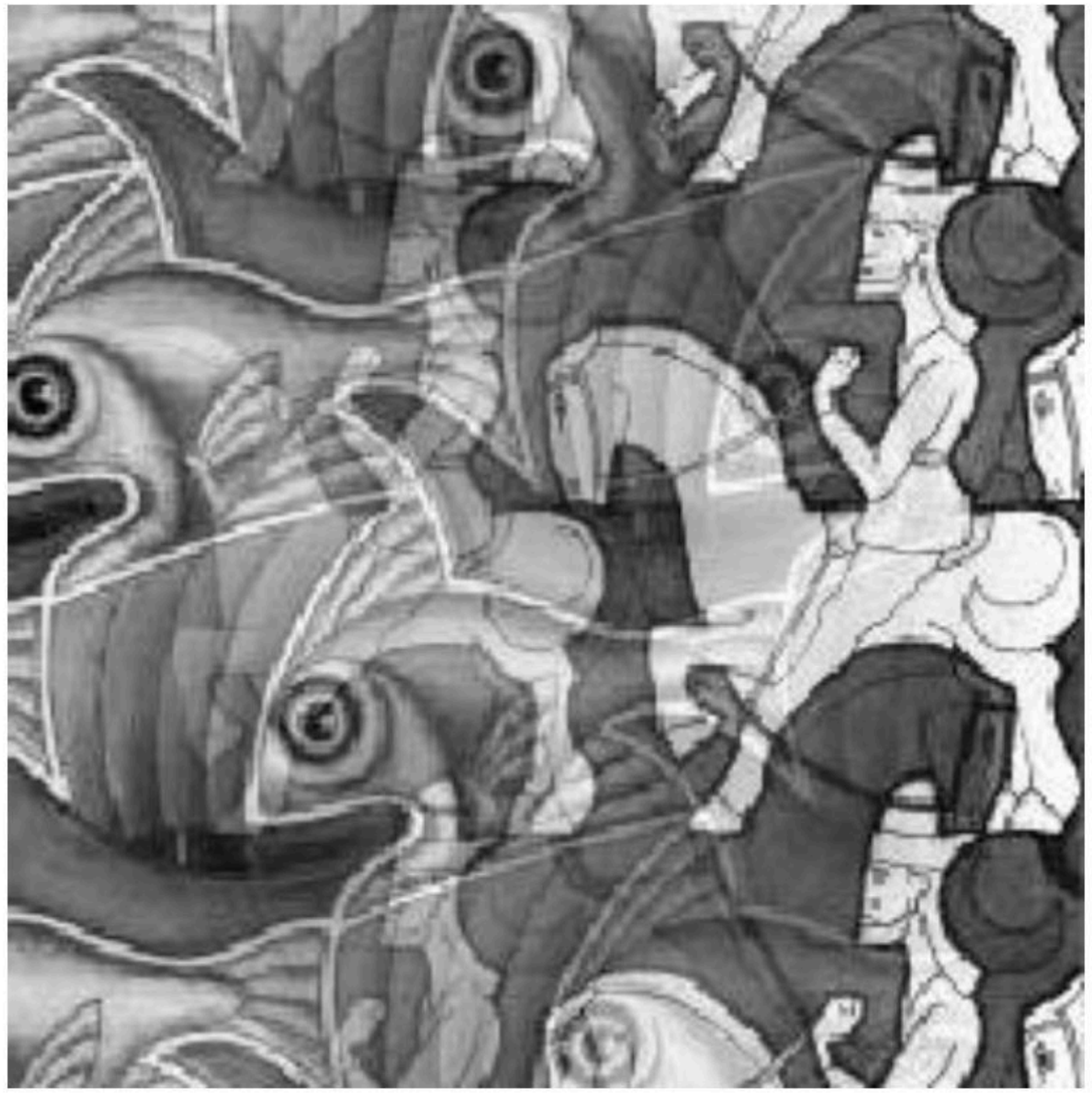
For a “smoother” look…

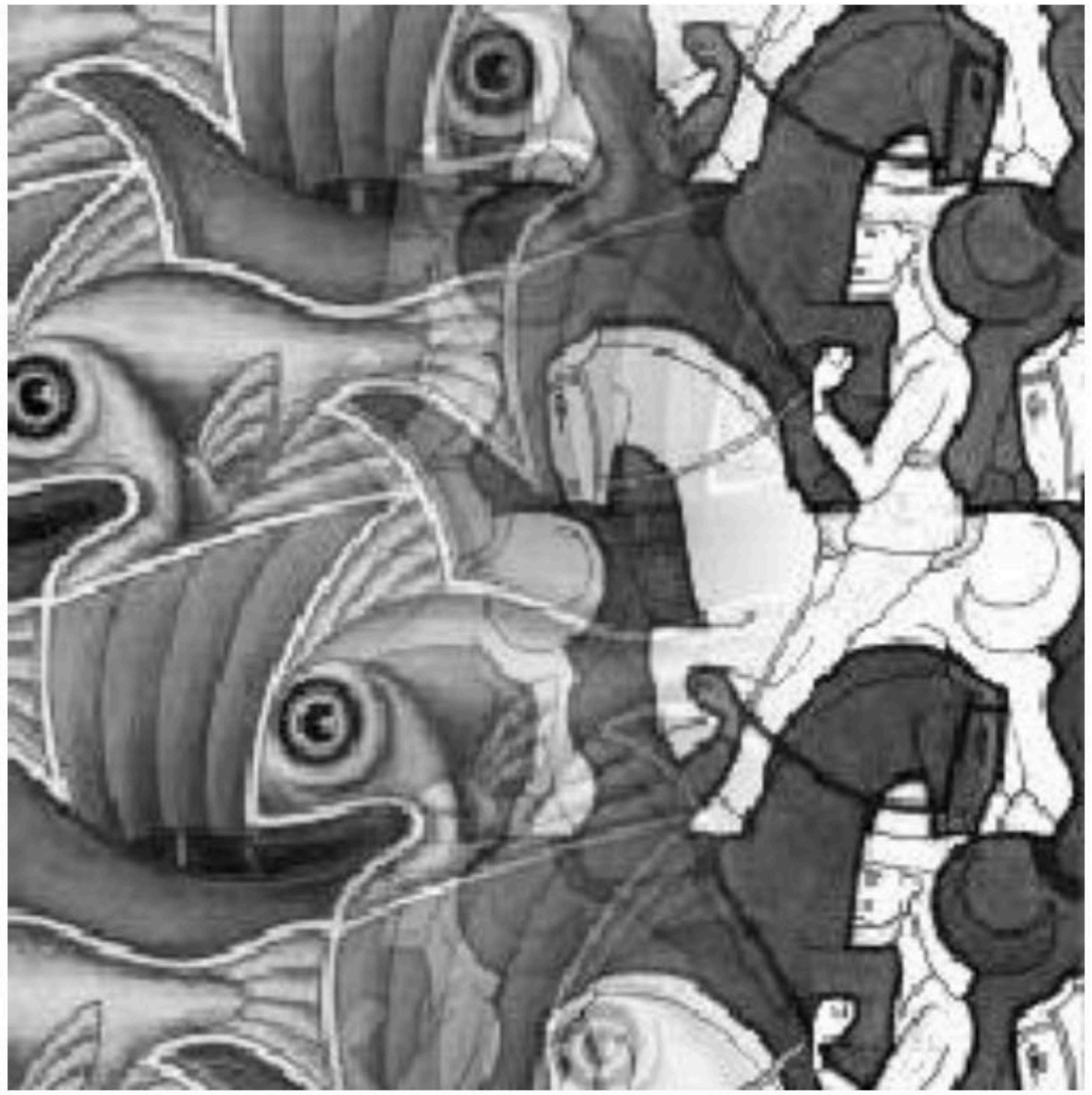


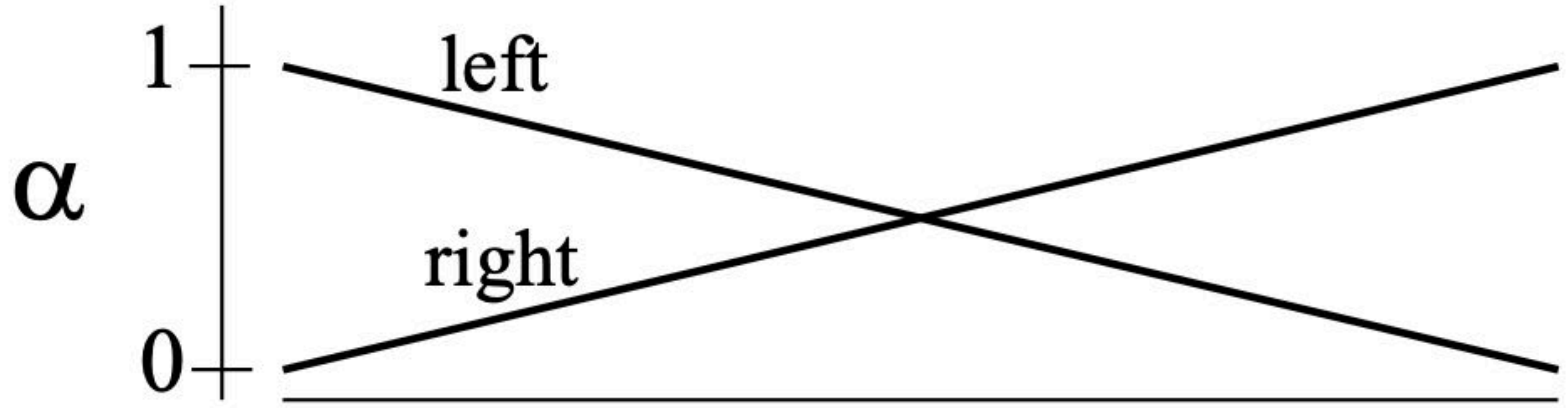
$$I_{\mathrm{blend}} = \alpha\, I_{\mathrm{left}} + (1-\alpha)\, I_{\mathrm{right}}$$

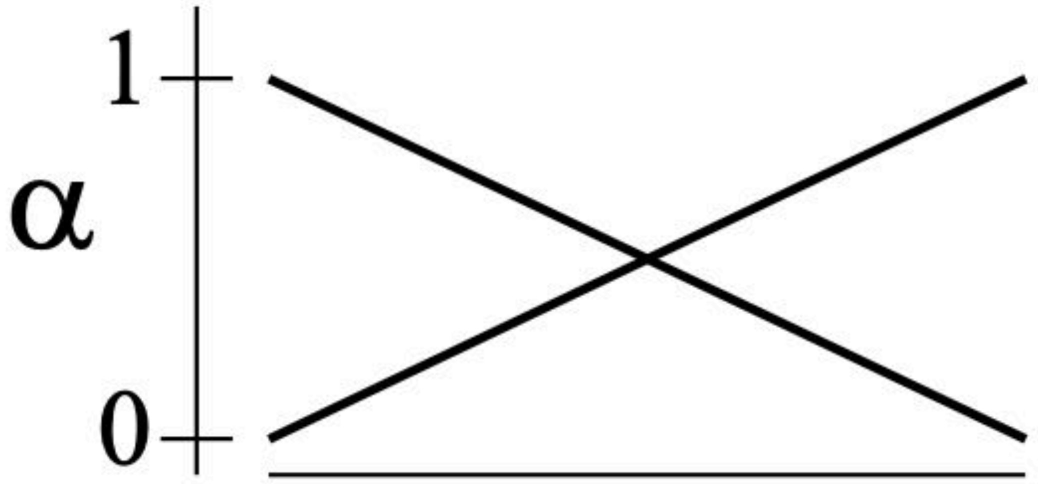
Slide credit: Efros

Effect of feather window size





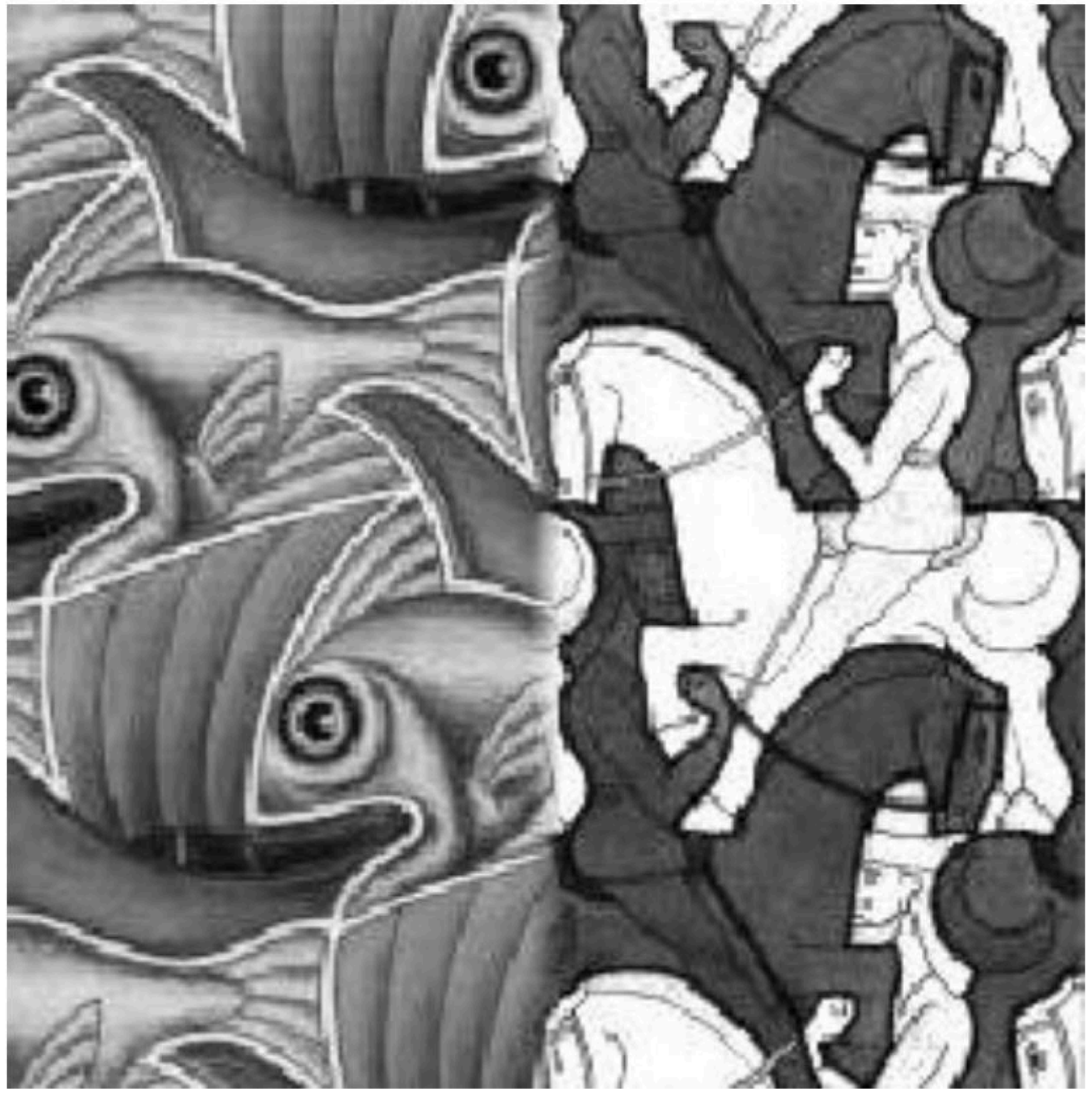


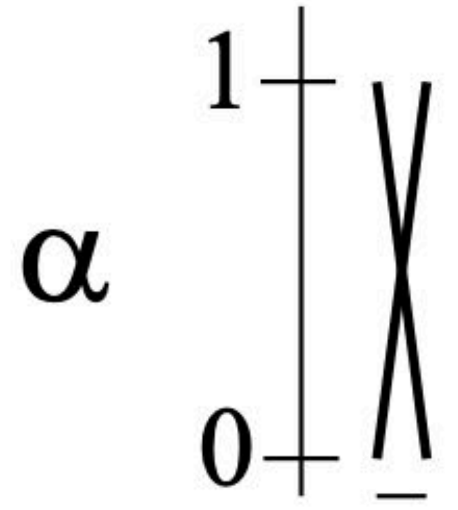


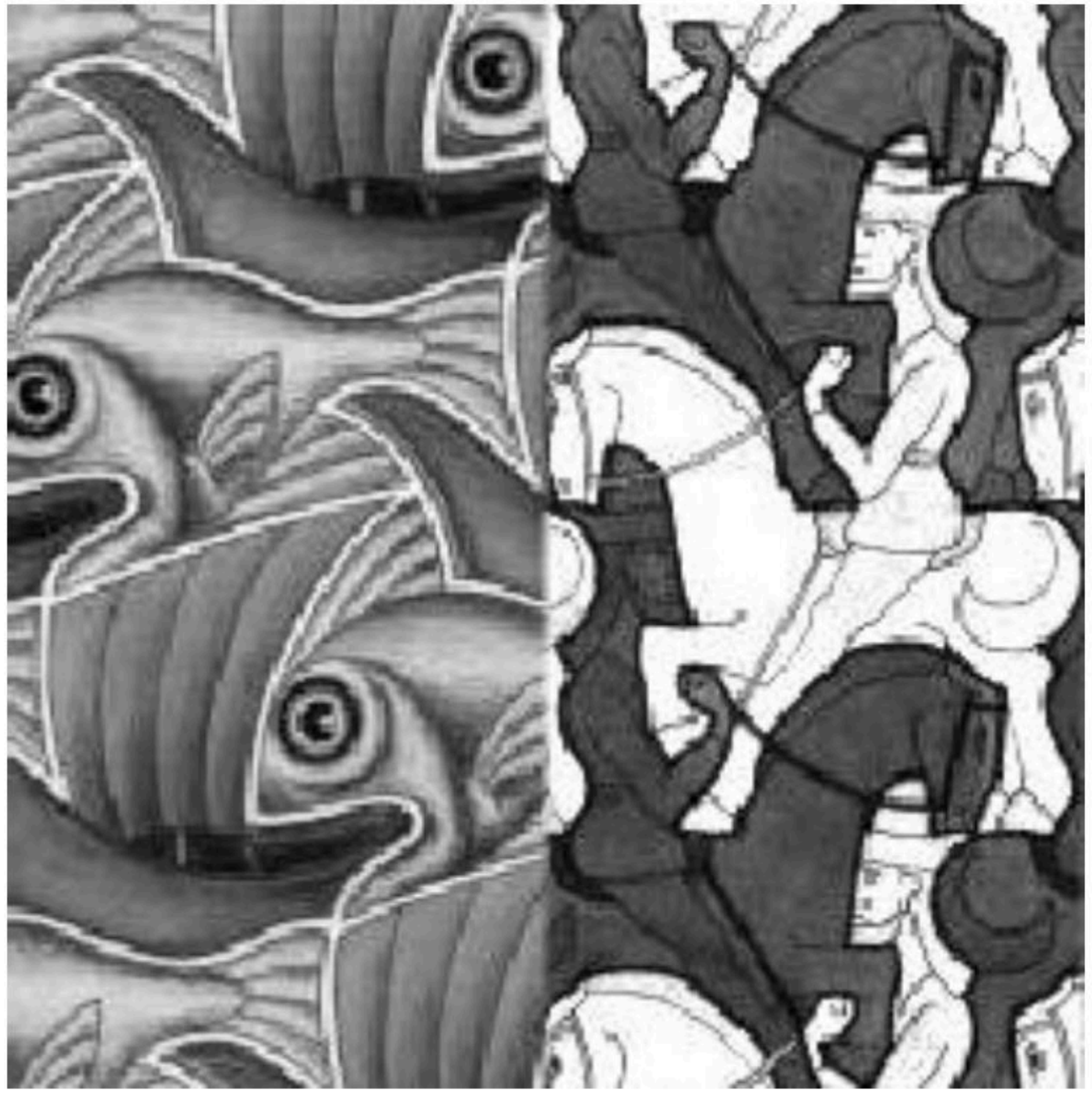
"Ghosting" visible is feather window (transition) is too large

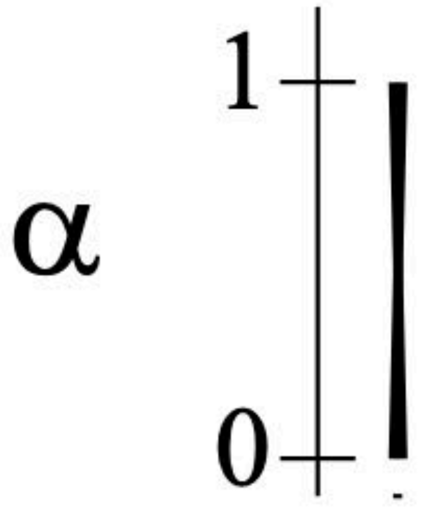
Slide credit: Efros

Effect of feather window size









Seams visible is feather window (transition) is too small

Slide credit: Efros

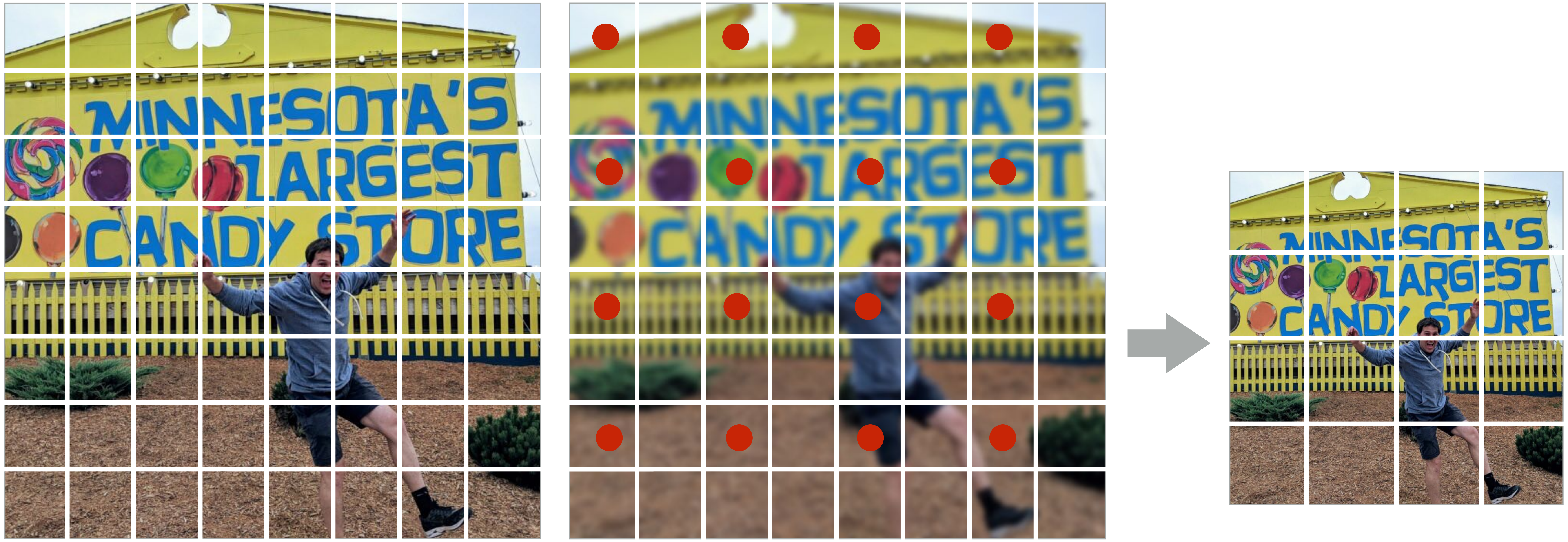
What do we want

- To avoid seams, transition window should be >= size of largest prominent feature
- To avoid ghosting, transition window should be smaller than ~ 2X smallest prominent feature
- In other words, the largest and smallest features need to be within a factor of two for feathering to generate good results
- Intuition:
  - Coarse structure of images (large features) should transition slowly between images
  - Fine structure should blend quickly!

Slide credit: Efros, Guerzhoy

Downsample

- Step 1: Remove high frequencies (aka blur)
- Step 2: Sparsely sample pixels (in this example: every other pixel)



Downsample

- **Step 1: Remove high frequencies (convolution)**
- **Step 2: Sparsely sample pixels (in this example: every other pixel)**

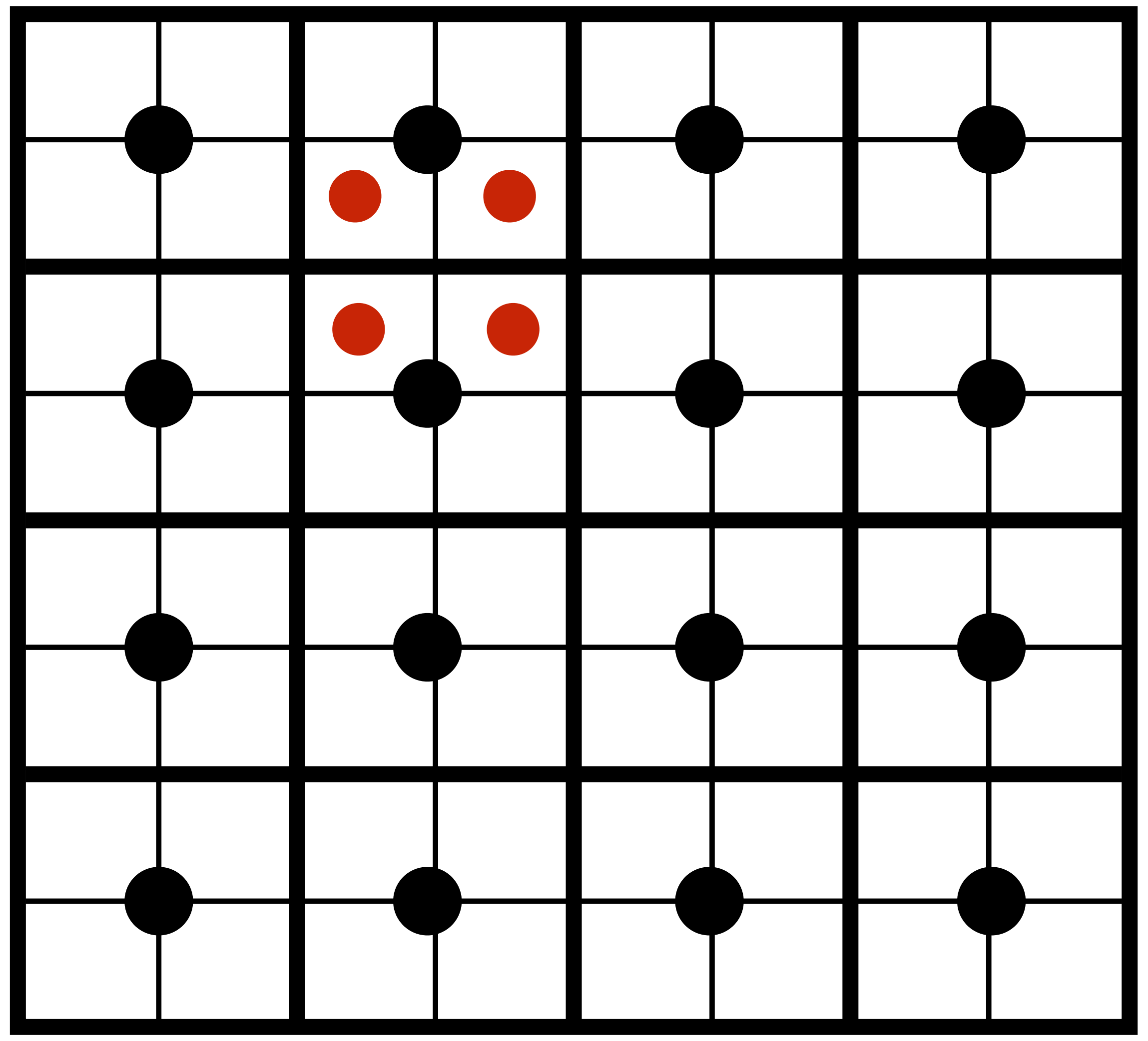
```
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH/2 * HEIGHT/2];

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/2; j++) {
   for (int i=0; i<WIDTH/2; i++) {
      float tmp = 0.f;
      for (int jj=0; jj<4; jj++)
         for (int ii=0; ii<4; ii++)
            tmp += input[(2*j+jj)*(WIDTH+2) + (2*i+ii)] * weights[jj*3 + ii];
      output[j*WIDTH/2 + i] = tmp;
  }
}
```

Upsample

Via bilinear interpolation of samples from low resolution image

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) ? .75f : .25f;
      float w2 = (j%2) ? .75f : .25f;

      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

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

G_0

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

Each (increasingly numbered) level in Laplacian pyramid represents a band of (increasingly lower) frequency information in the image

[Burt and Adelson 83]

Laplacian pyramid

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

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

Laplacian pyramid

$L_0 = G_0 - up(G_1)$

$L_1 = G_1 - up(G_2)$

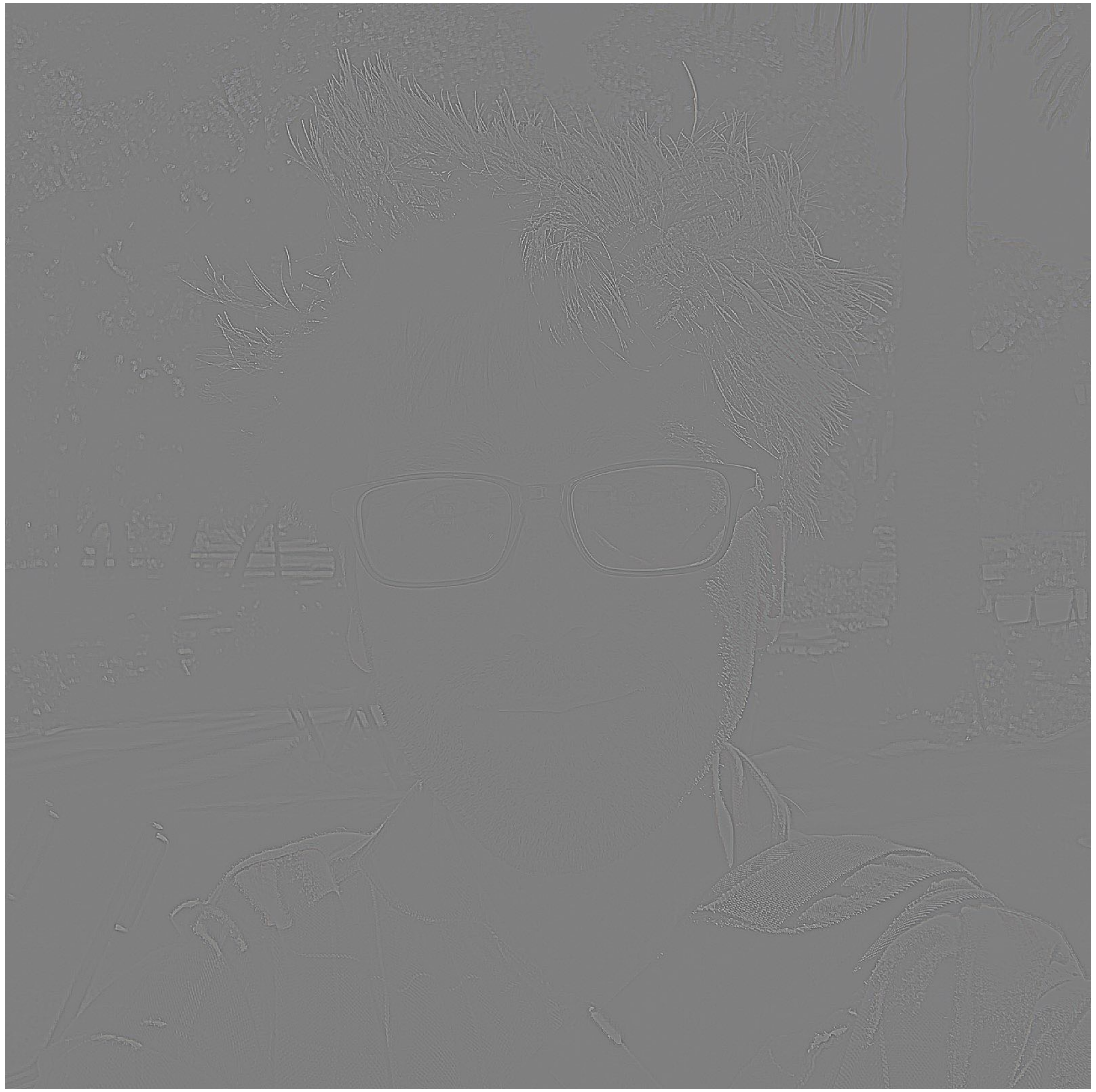
$L_2 = G_2 - up(G_3)$

$L_3 = G_3 - up(G_4)$

$L_4 = G_4$

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

Laplacian pyramid



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

Laplacian pyramid

$L_1 = G_1 - up(G_2)$

Laplacian pyramid

$L_2 = G_2 - up(G_3)$

Laplacian pyramid

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

Laplacian pyramid

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

Laplacian pyramid

$L_5 = G_5$

Gaussian/Laplacian pyramid summary

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

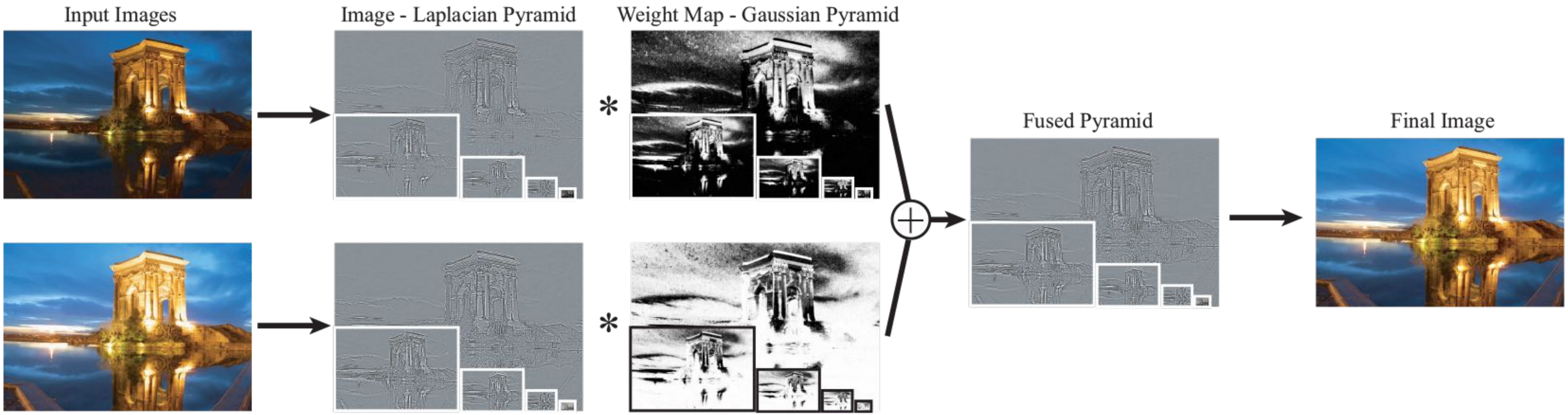
- $G_i(x,y)$ — frequencies up to limit given by *i*

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

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

Use of Laplacian pyramid in local tone mapping

- Compute weights for all Laplacian pyramid levels
- Merge pyramids (image features) not image pixels
- Then "flatten" merged pyramid to get final image



Merging Laplacian pyramids

Four exposures (weights not shown)

Merged result
(after blurring weight mask)
Notice "halos" near edges

Merged result
(based on multi-resolution pyramid merge)

Why does merging Laplacian pyramids work better than merging image pixels?

Summary: simplified image processing pipeline

- Correct pixel defects
- Align and merge (to create high signal to noise ration RAW image)
- Correct for sensor bias (using measurements of optically black pixels)
- Vignetting compensation
- White balance

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

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

3x10 bits per pixel
RGB intensity per pixel
Pixel values linear in energy

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

3x8-bits per pixel
Pixel values perceptually linear

Acknowledgements

- Thanks and credit for slides to Ren Ng and Marc Levoy