Resampling and Image Pyramids

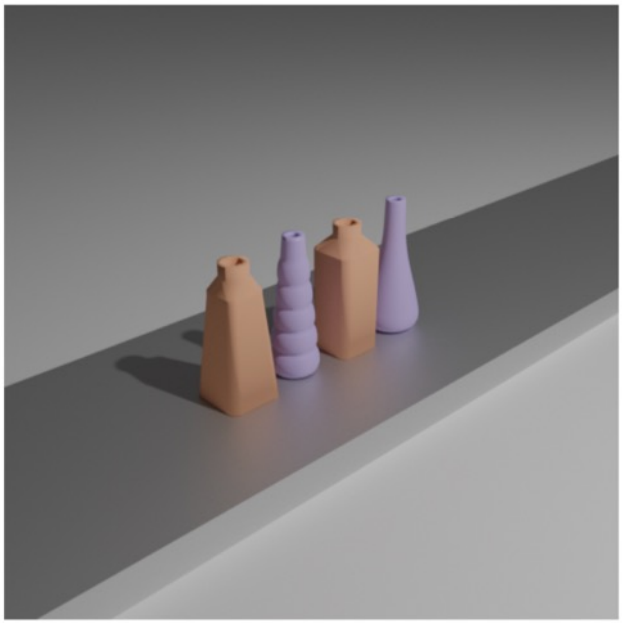
CS 482, Prof. Stein

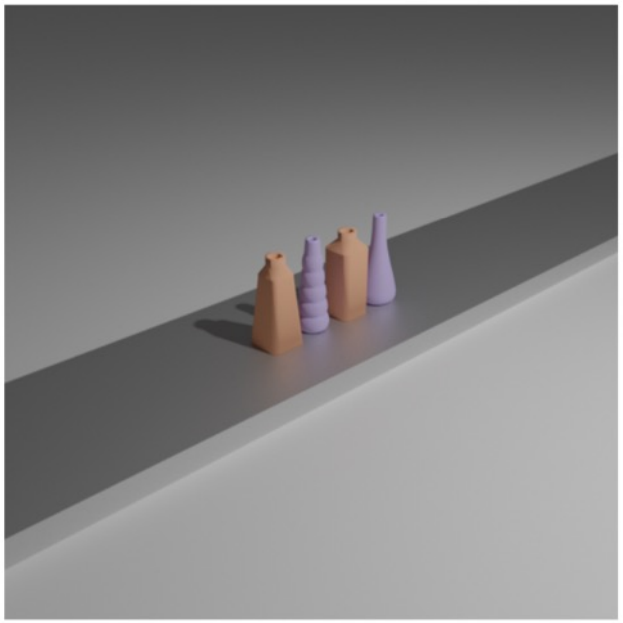
Lecture 04

Motivation: Resizing images is surprisingly important

Motivation: Resizing images is surprisingly important

What if we want to find objects common to two images:

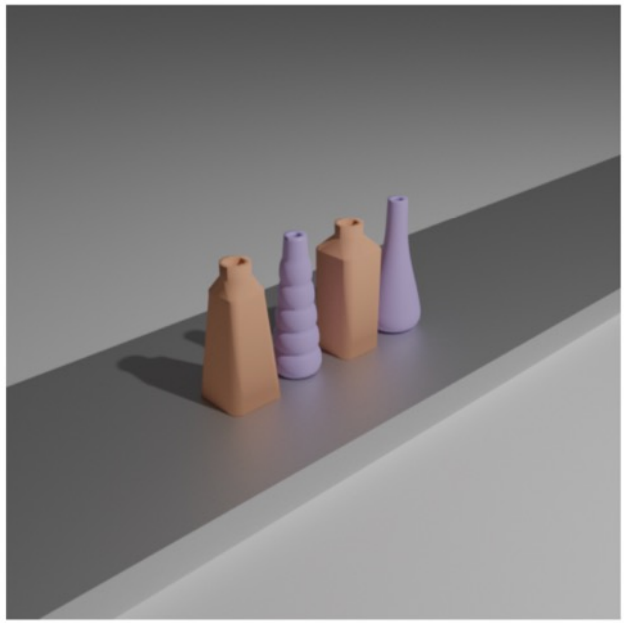


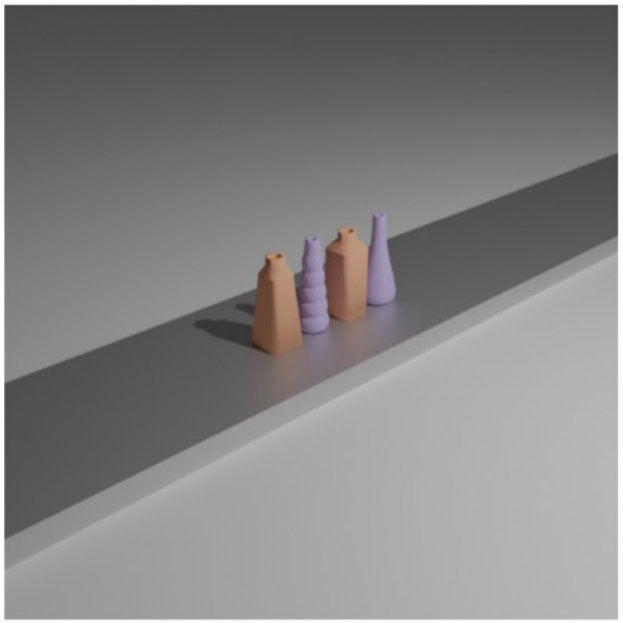


How might we "match" them computationally?

Motivation: Resizing images is surprisingly important

What if we want to find objects common to two images:



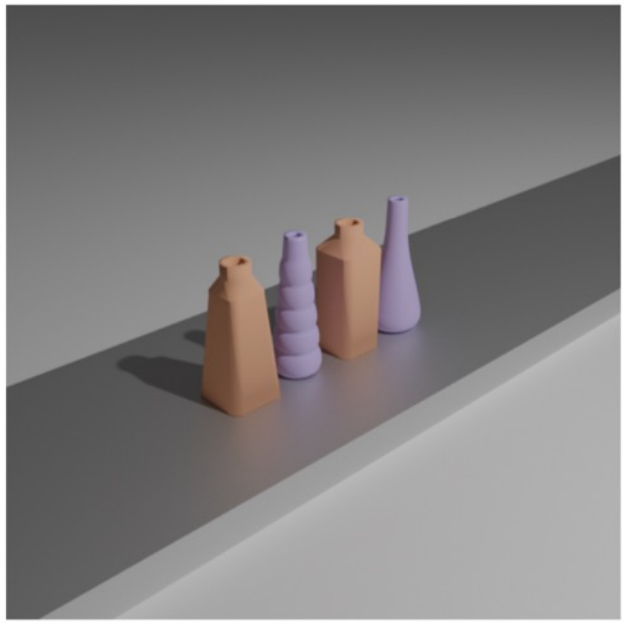


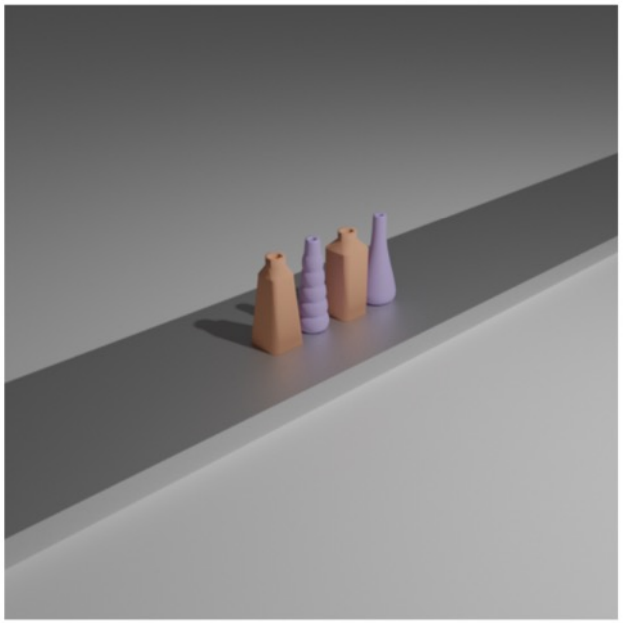
What if we take the derivative?

What if we use curvature?

Motivation: Resizing images is surprisingly important

What if we want to find objects common to two images:





What if we take the derivative?

What if we use curvature?

Neither of these properties are *scale invariant.*

Handling *scale* is a challenging problem in image-space. It is often easier to simply resize the images themselves.

Motivation: How do we resize an image?

Today we'll be talking about how to change the size *of an image, and the choices we need to make when estimating values* between *pixels.*

Reading & Slide Credits

Readings:

- Szeliski 3.5

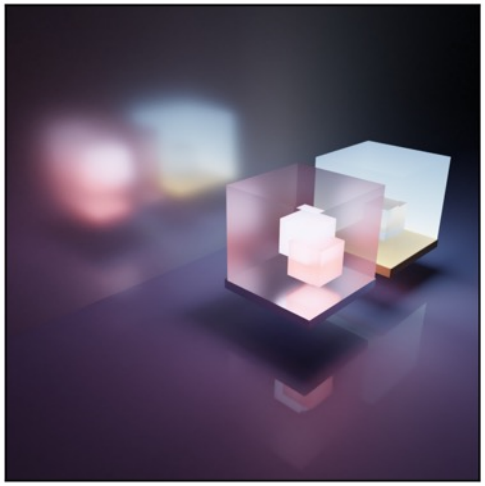
Slide Credits (from which many of these slides are either directly taken or adapted)

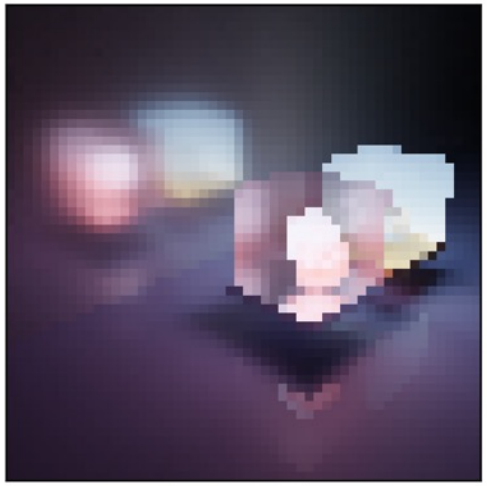
- CMU Computer Vision Course (Yannis Gkioulekas)
- Cornell Tech Computer Vision Course (Noah Snavely)
- Toronto CV Course

Downsizing Images & Image Pyramids

What's the simplest way to downsample an image?

What if we remove every N'th row and column?

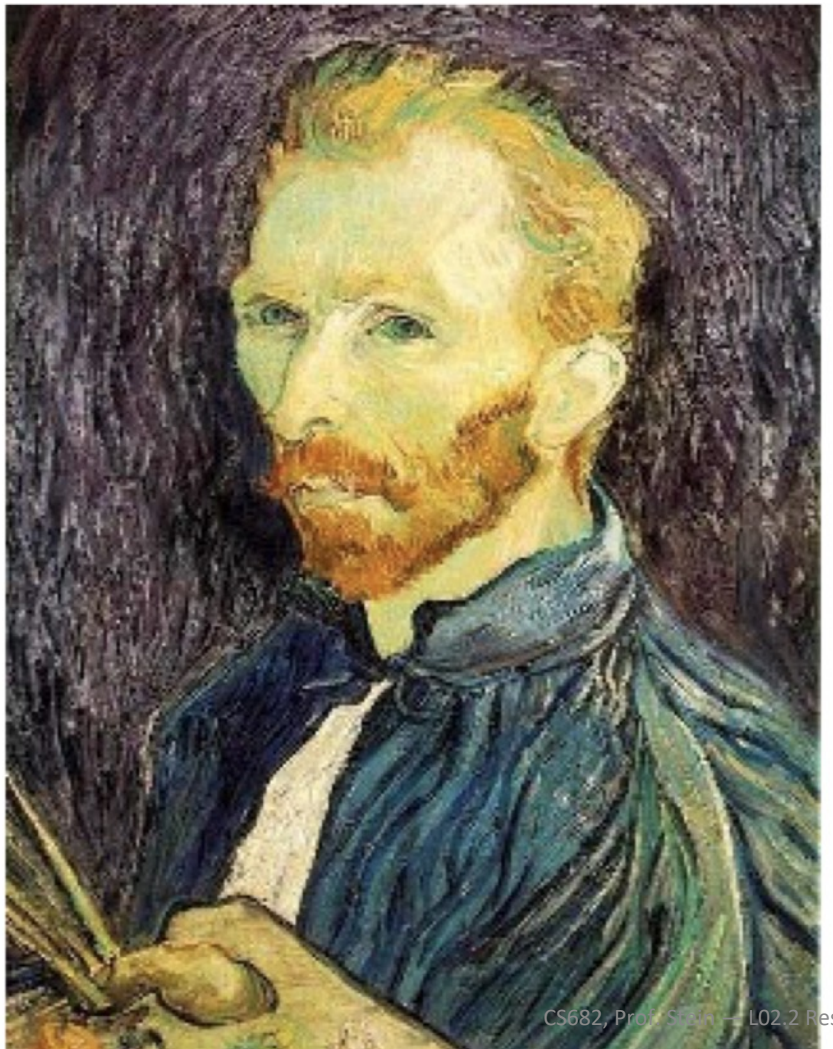




This is called *Image Sub-Sampling*

So are we done? What's the problem?

Problem: Naïve downsampling can result in artifacts.



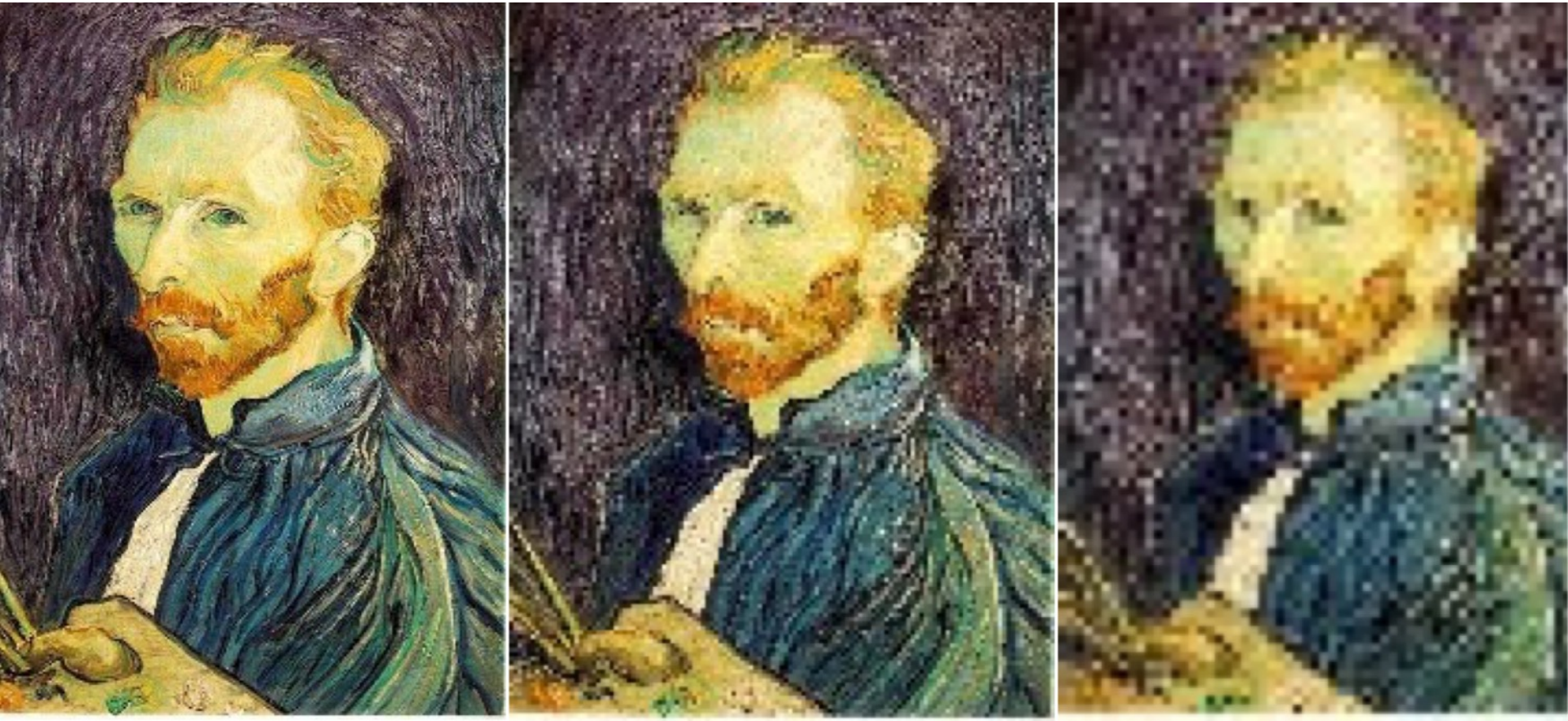
The smaller images look a bit grainy? Why?

1/4

1/8

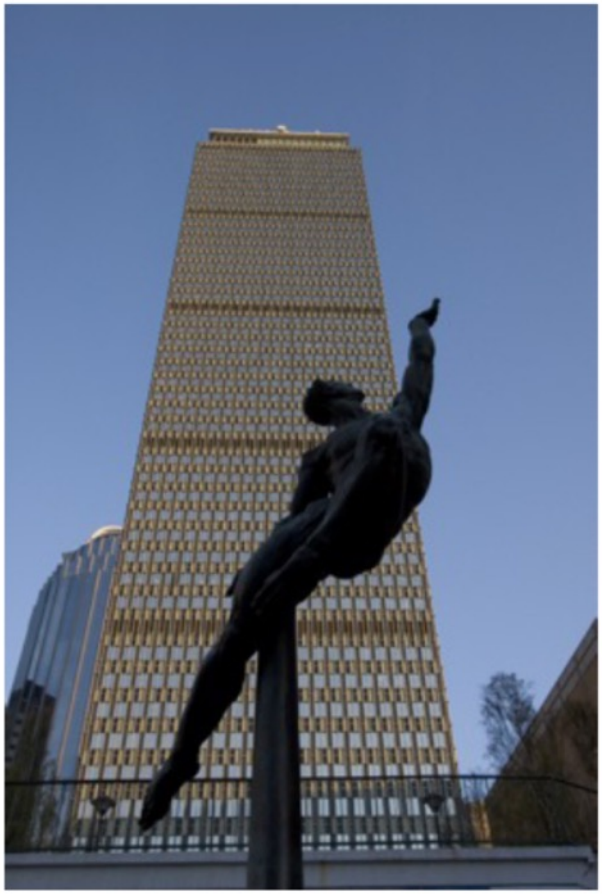
Problem: Naïve downsampling can result in artifacts.

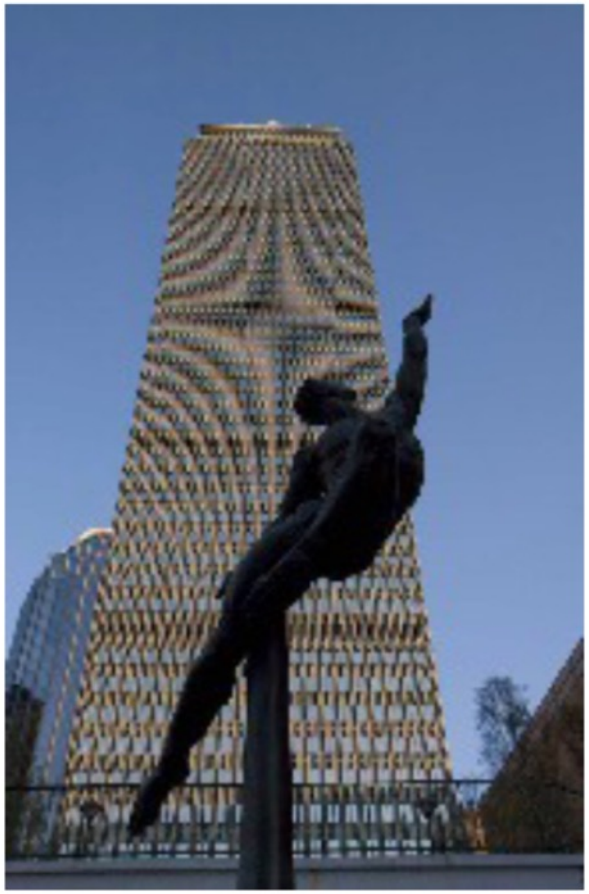
Here are zoomed in versions of the same images:



1/2 1/4 (2x zoom) 1/8 (4x zoom)

Problem: Naïve downsampling can result in artifacts.



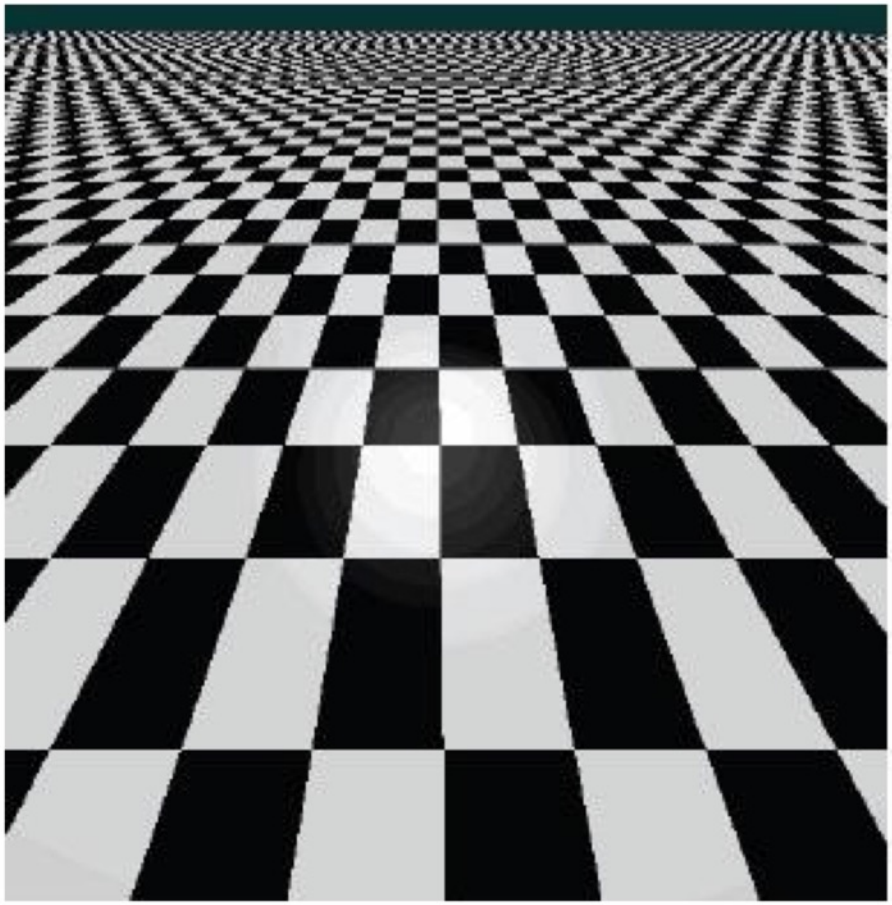


This type of artifact is known as a *moiré pattern.*

Problem: Naïve downsampling can result in artifacts.

This is *aliasing*, like we saw in the context of Fourier Transforms

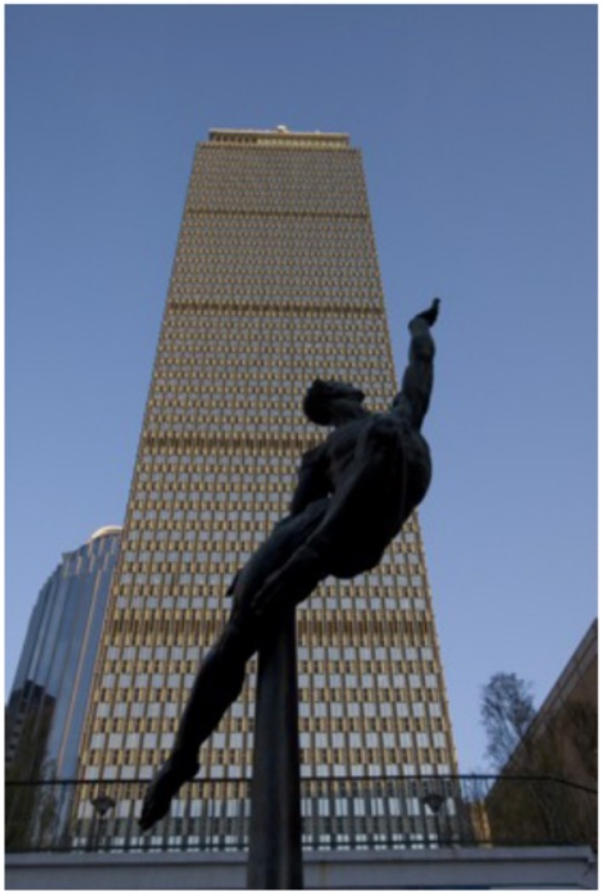
(remember the Wagon Wheel effect?)

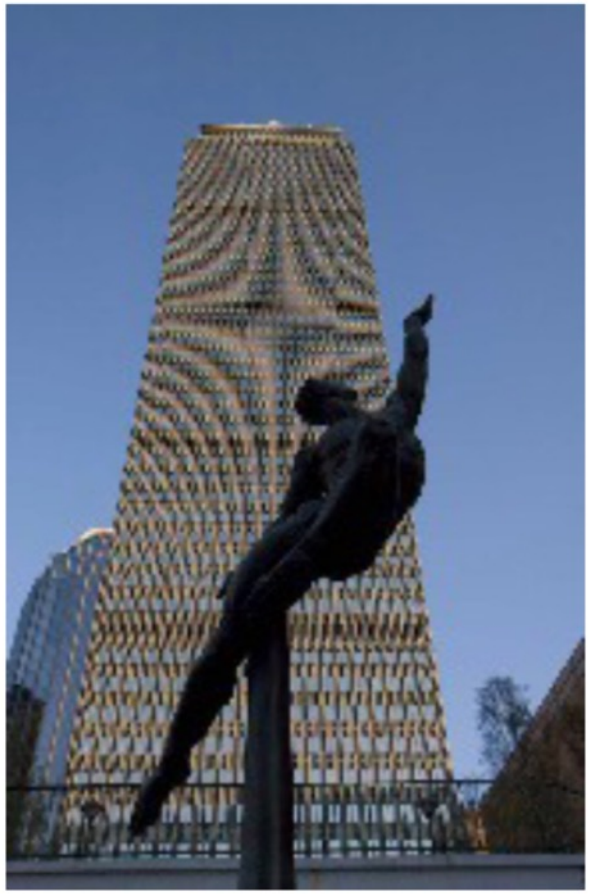


Artifacts are also very common in computer graphics.

Synthetic images are subject to all sorts of issues…

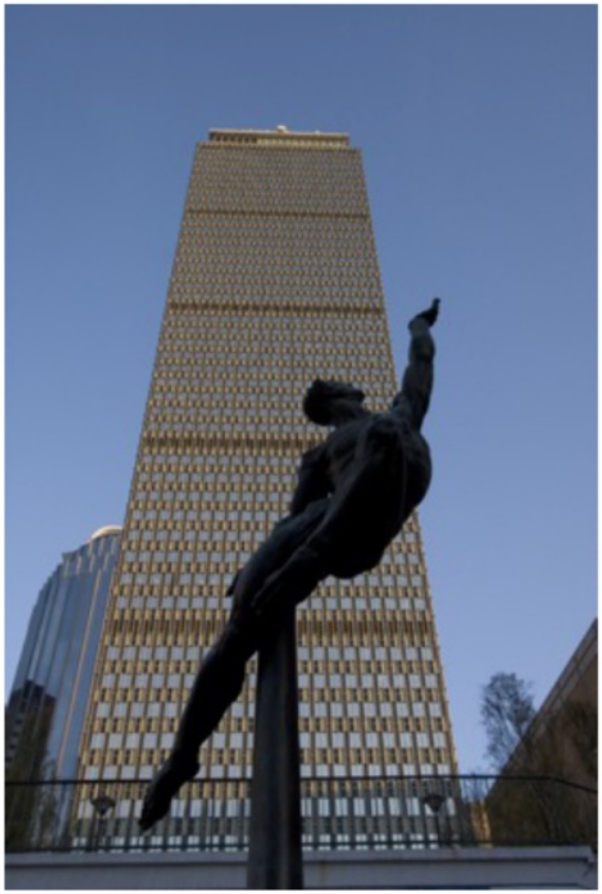
So why do these artifacts occur when downsampling but not in the original?

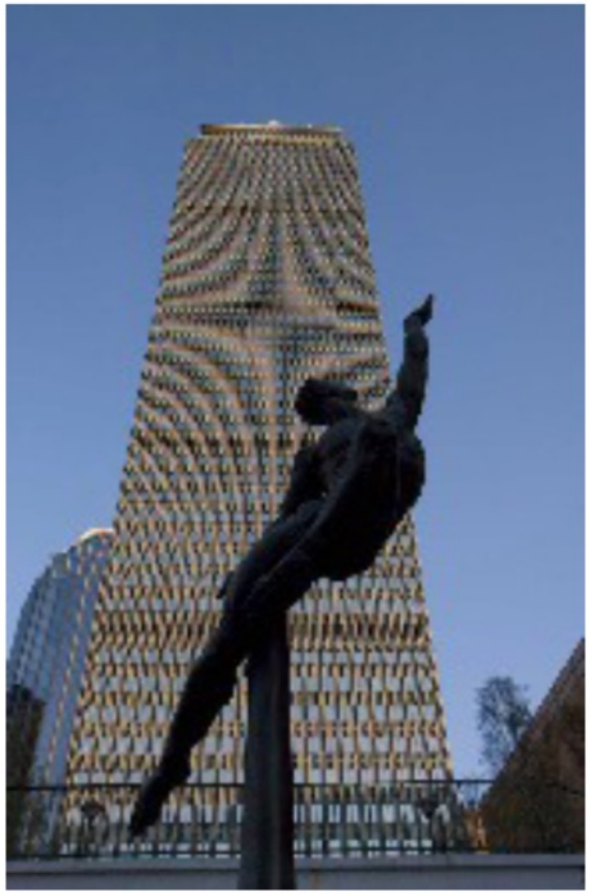




So why do these artifacts occur when downsampling but not in the original?

Question: if the smaller image were taken by a camera, would we have the same effect?

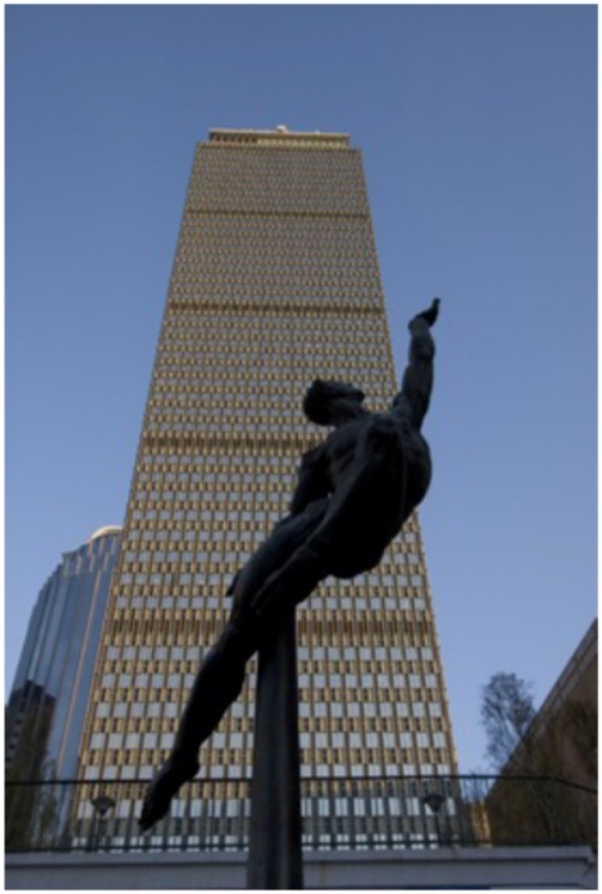


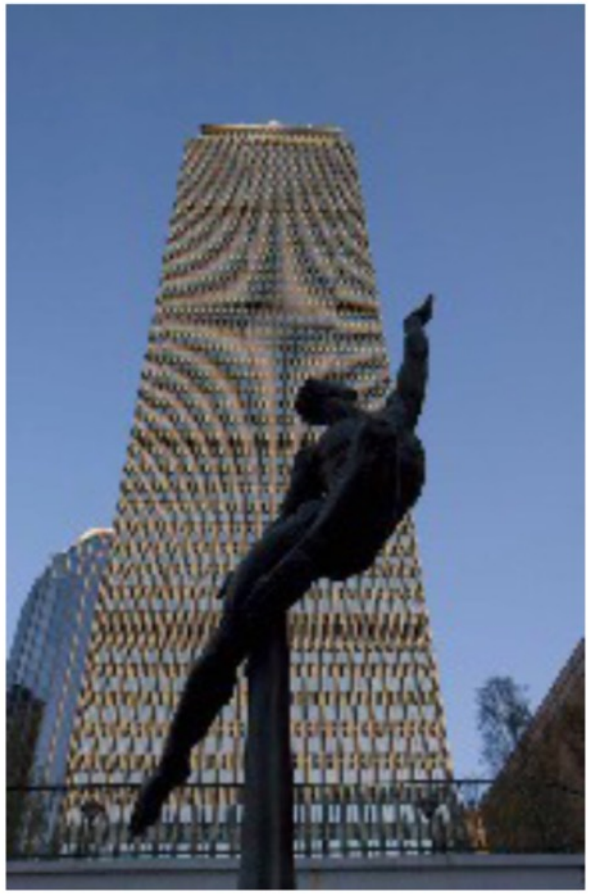


So why do these artifacts occur when downsampling but not in the original?

Question: if the smaller image were taken by a camera, would we have the same effect?

No! (or … probably not) because the light hitting each pixel on the sensor would be averaged together during collection.



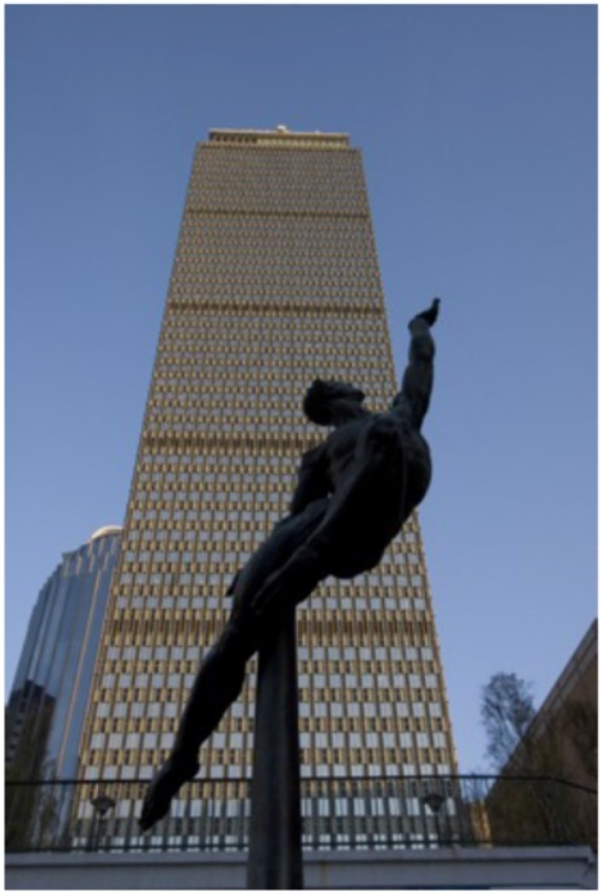


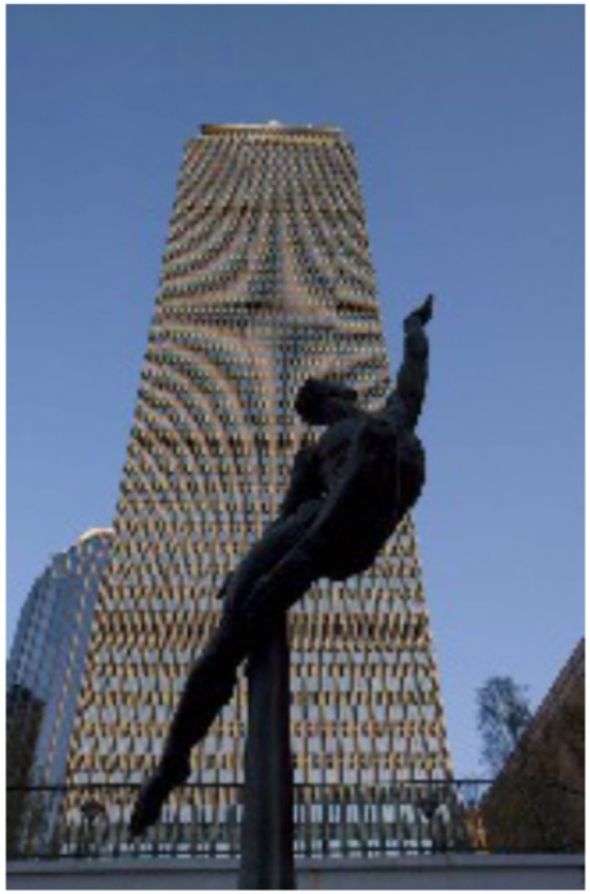
So why do these artifacts occur when downsampling but not in the original?

Question: if the smaller image were taken by a camera, would we have the same effect?

No! (or … probably not) because the light hitting each pixel on the sensor would be averaged together during collection.

Let's try blurring before downsampling, to mimic this effect.



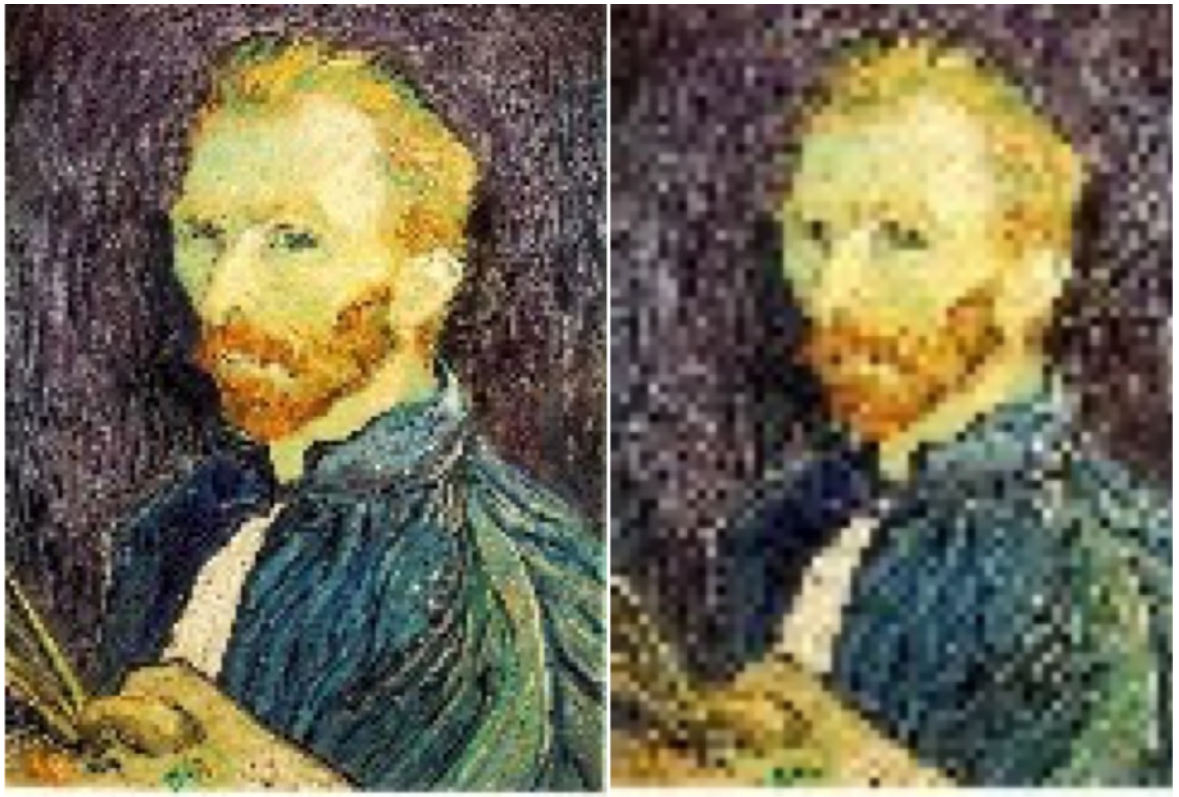


So why do these artifacts occur when downsampling but not in the original?

Question: if the smaller image were taken by a camera, would we have the same effect?

No! (or … probably not) because the light hitting each pixel on the sensor would be averaged together during collection.

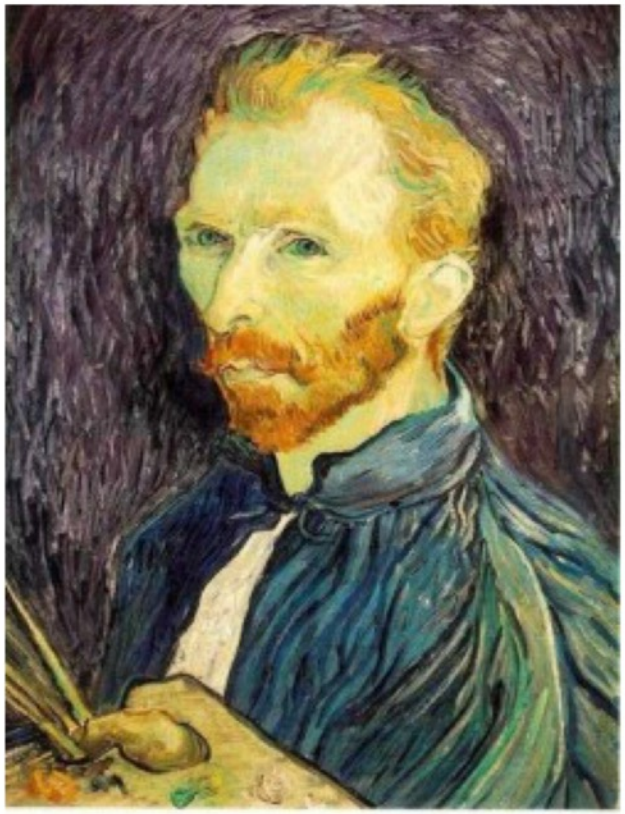
Let's try blurring before downsampling, to mimic this effect.



1/4 (2x zoom) 1/8 (4x zoom)

This is related to the Nyquist Limit: the the artifacts come from high-frequency terms that can no longer be represented.

Using G*aussian Prefiltering*, the images look much "nicer" after downsampling



Gaussian 1/2

We alternately filter and subsample the image.

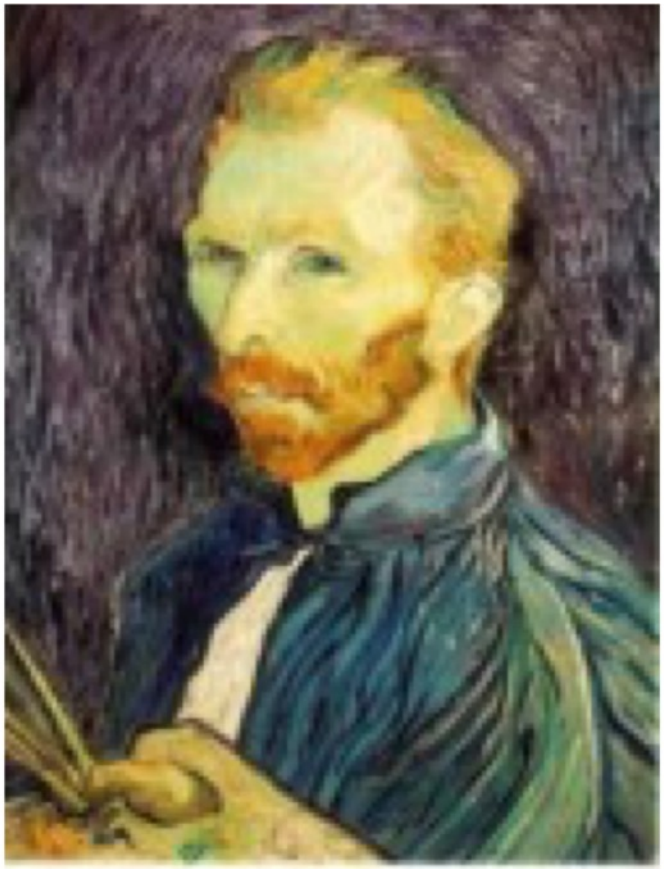
G 1/4

G 1/8

Notice that those artifacts are effectively gone.

Let's compare the two techniques: First, with Gaussian Prefiltering

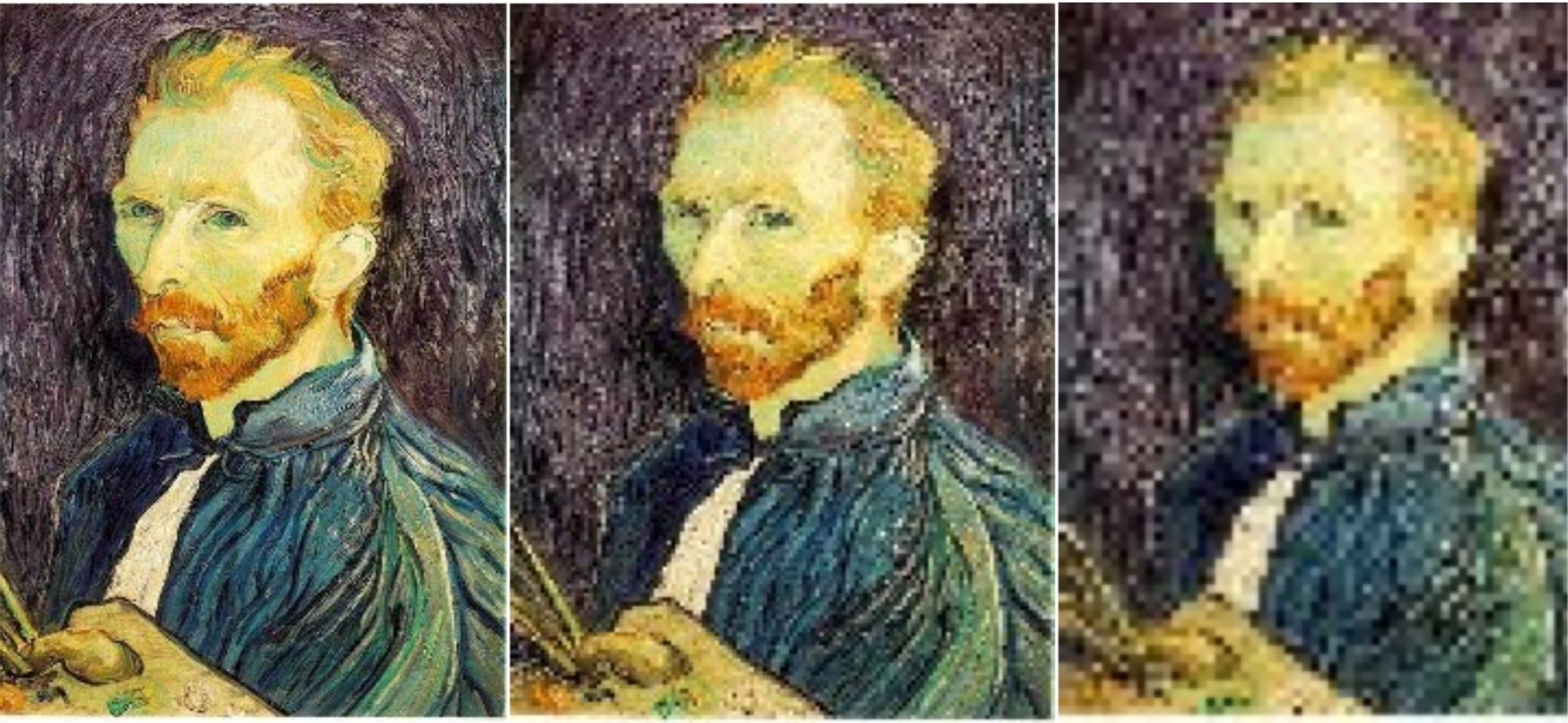
Gaussian 1/2



G 1/4

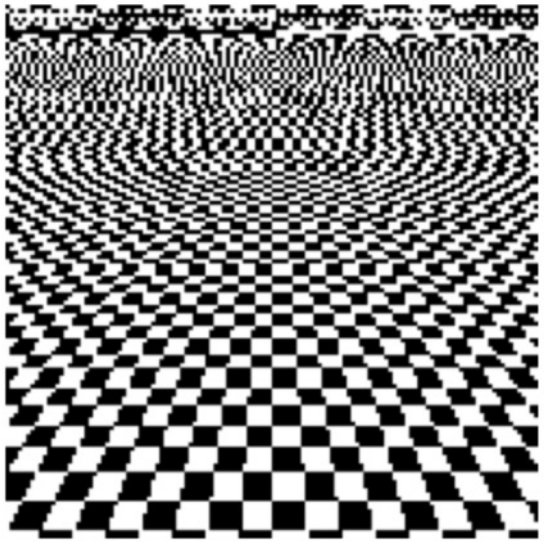
G 1/8

Let's compare the two techniques:
Next, without Gaussian Prefiltering (naïve)

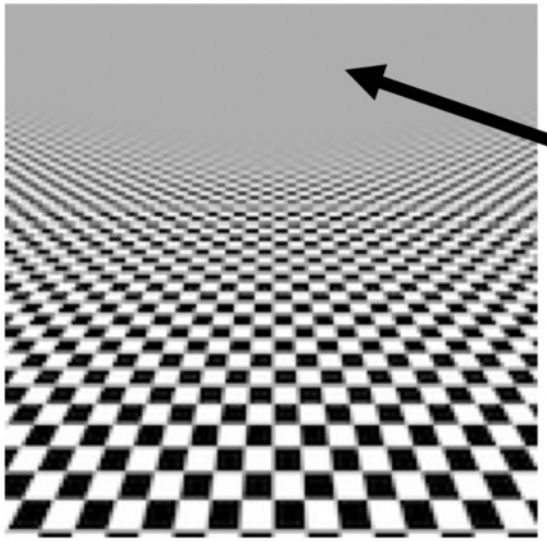


1/2 1/4 (2x zoom) 1/8 (4x zoom)

What happens when we blur and downsample the checkerboard pattern?



Naïve subsampling



Proper prefiltering
("antialiasing")

Image turns grey!
(Average of black and white squares, because each pixel contains both.)

This is a very computer graphics solution: if I want an antialiased image, I should make a bigger image and then blur and downsample it.

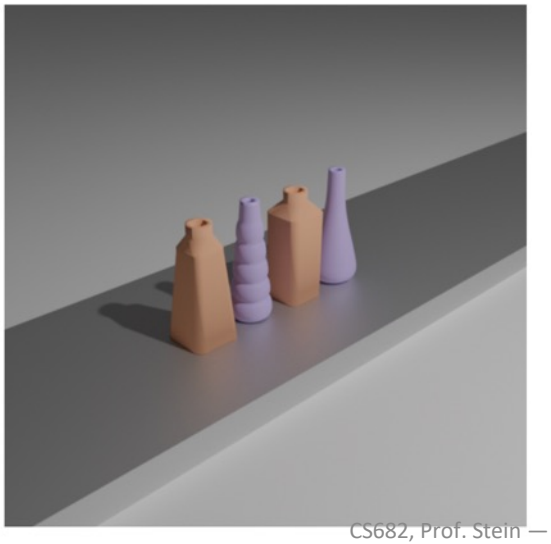
So how do we know what size the image should be?

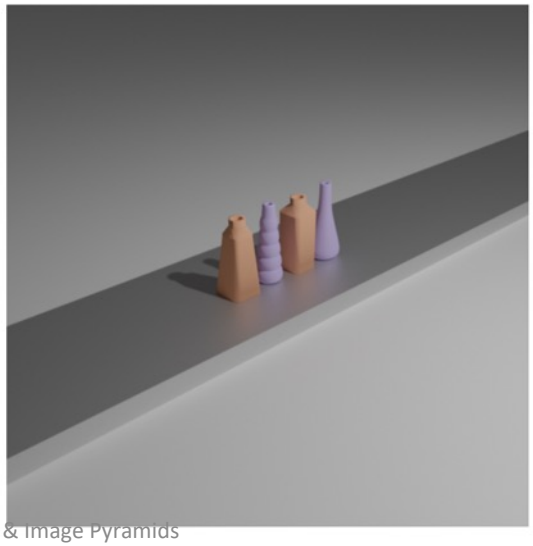
So how do we know what size the image should be?

To be clear, this depends strongly on the application.

In the previous section, we assumed that we were told how big the input image was and how big it should be.

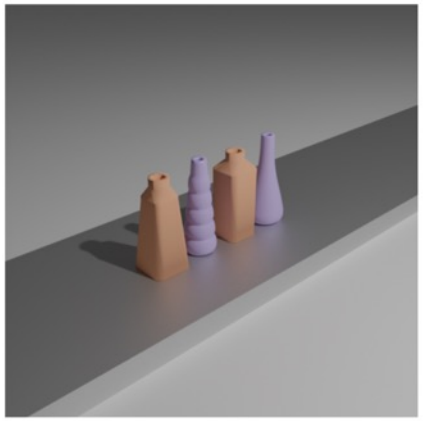
What if we aren't told how big the image needs to be?

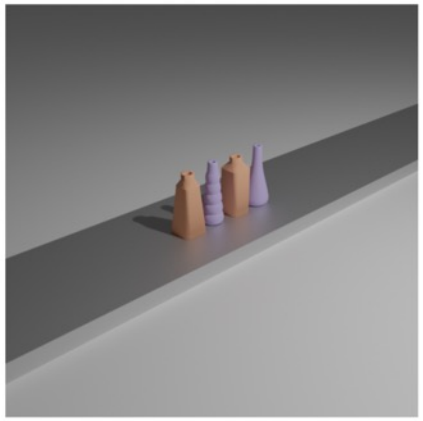




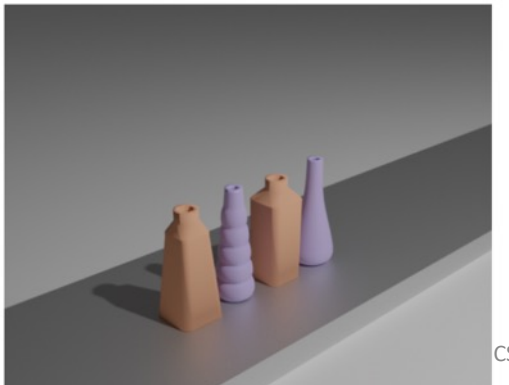
So how do we know what size the image should be?

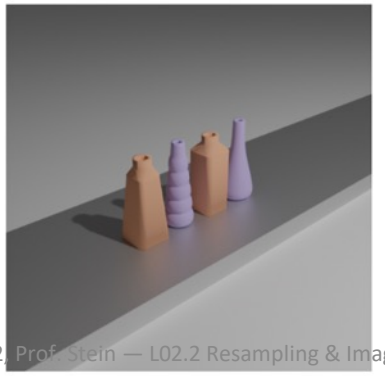
Let's revisit our object matching example?

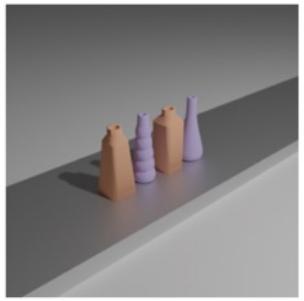




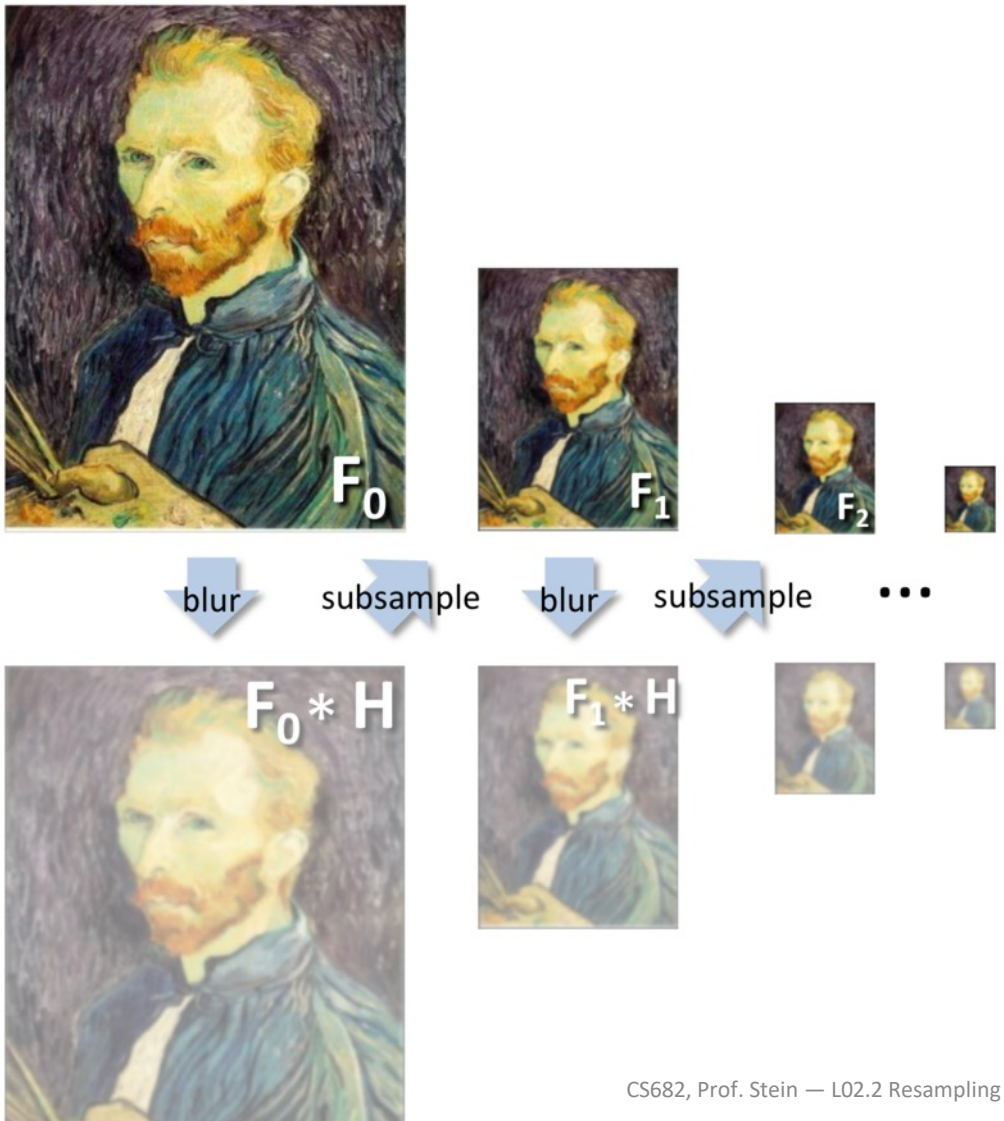
Let's just compute multiple image scales, and *hope* that one of the resized images is at the scale we want.



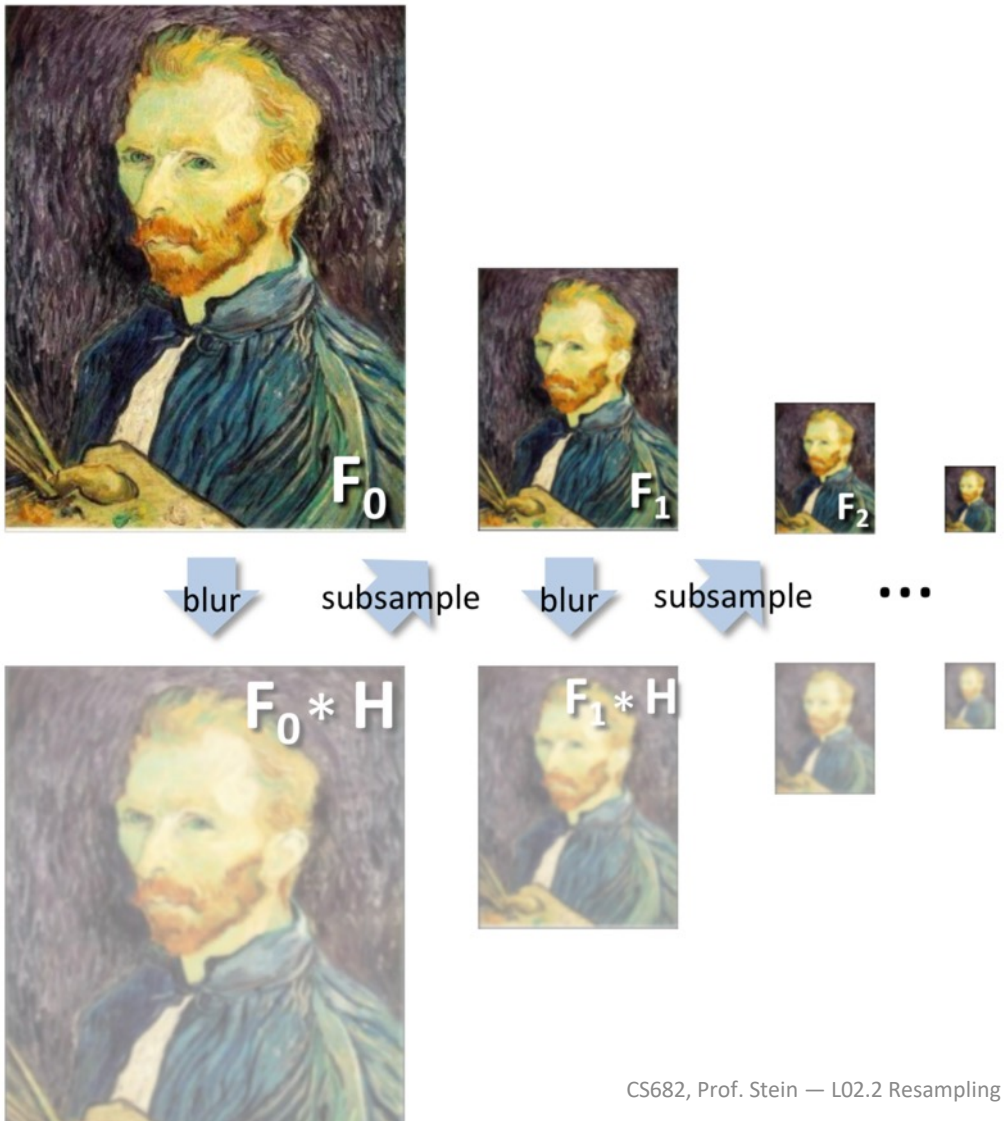




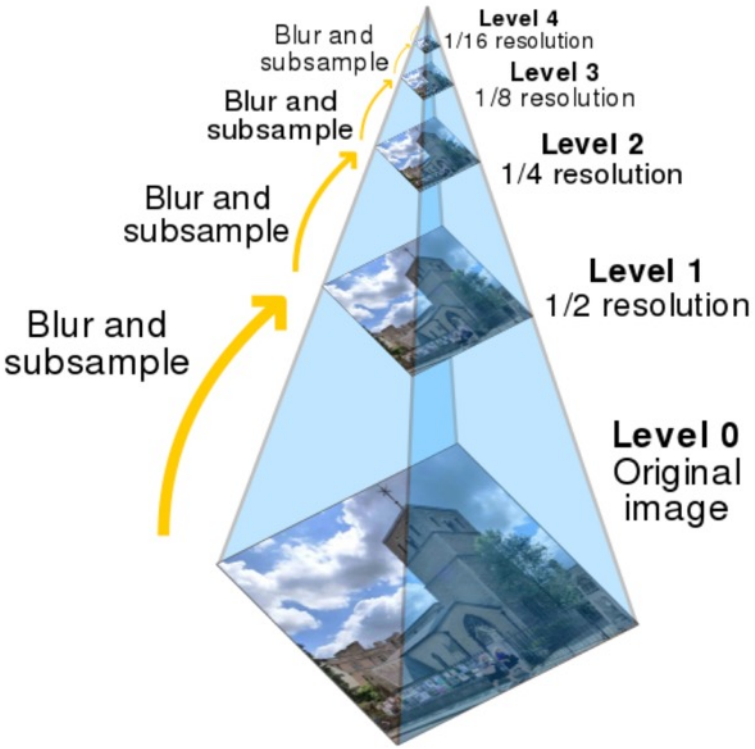
This sequence of images (blurred and subsampled) is known as a *Gaussian Pyramid*



The sequence of images (blurred and subsampled) is known as a *Gaussian Pyramid*



In this pyramid, we store all of the images as a sequence.



The sequence of images (blurred and subsampled) is known as a *Gaussian Pyramid*

Input: Image I of size $(2^N+1)x(2^N+1)$

Output: Images $g_0, g_1, \ldots, g_{N-1}$

where the size of g_i is:

$(2^{N-i}+1)x(2^{N-i}+1)$

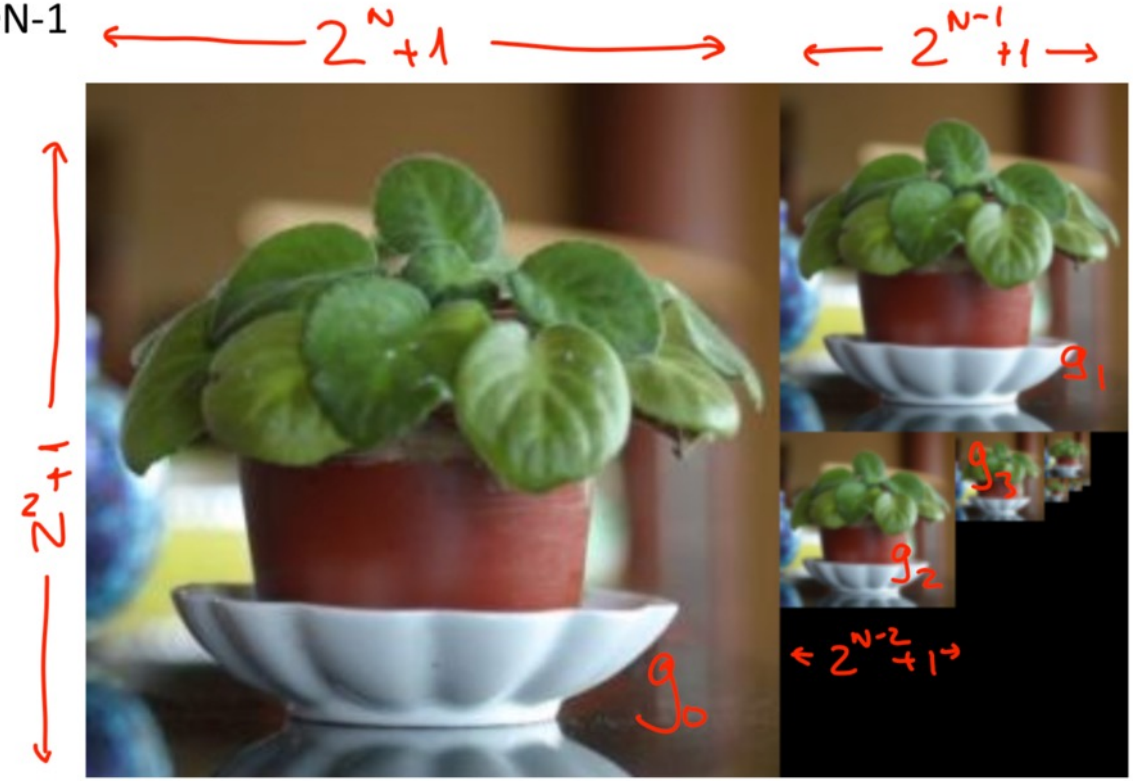
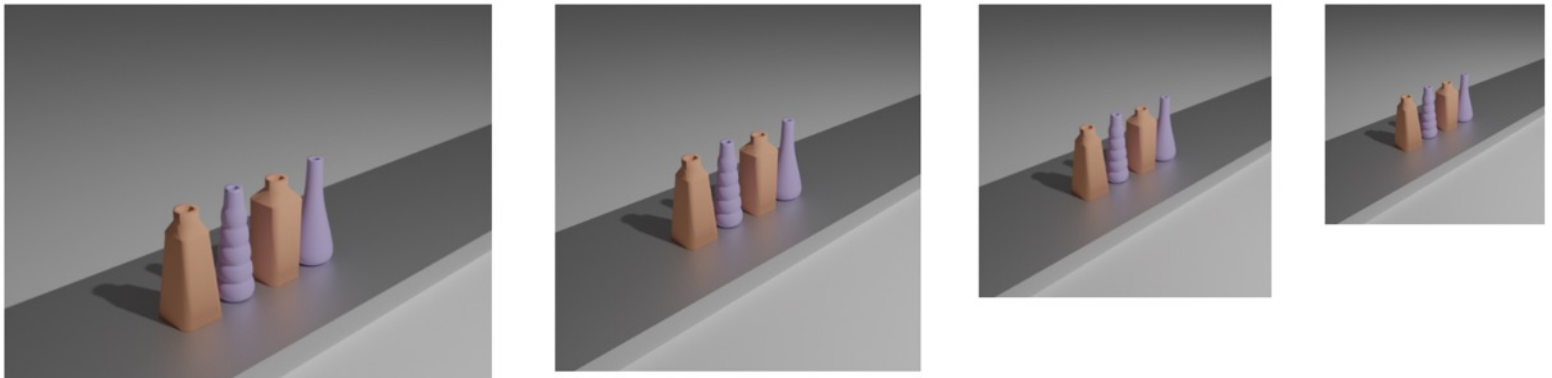


Image Pyramids will come up a number of times later on in the course

It can be quite useful to have the same image at multiple scales.

Particularly when we are talking about *feature detection*, we will be looking for corners or blobs at multiple image scales so that we can find objects from different locations.



So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric:

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric: $k = \begin{bmatrix} c & b & a & b & c \end{bmatrix}$

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric: $k = \begin{bmatrix} c & b & a & b & c \end{bmatrix}$

Constant images should remain unchanged:

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric: $k = \begin{bmatrix} c & b & a & b & c \end{bmatrix}$

Constant images should remain unchanged: $a + 2b + 2c = 1$

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric: $k = \begin{bmatrix} c & b & a & b & c \end{bmatrix}$

Constant images should remain unchanged: $a + 2b + 2c = 1$

Each pixel should contribute equally:

(This is called, unsurprisingly, the *Equal Contribution Criterion)*

So how do we pick the kernel for the Gaussian Pyramid?

The basic idea is built upon simple ideas, but how do we pick the kernel? Does it even *need* to be Gaussian?

Let's say our (1D, 5-element) kernel looks like this:

$$k = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

We want the kernel to be symmetric: $k = \begin{bmatrix} c & b & a & b & c \end{bmatrix}$

Constant images should remain unchanged: $a + 2b + 2c = 1$

Each pixel should contribute equally: $a + 2c = 2b = 1/2$

(This is called, unsurprisingly, the *Equal Contribution Criterion)*

So how do we pick the kernel for the Gaussian Pyramid?

Combined, our guidelines result in the following (parametric) kernel:

$$k = \begin{bmatrix} c \\ b \\ a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1/4 - a/2 \\ 1/4 \\ a \\ 1/4 \\ 1/4 - a/2 \end{bmatrix}$$

A few things to notice:

- The 2D version is the outer product of two 1D kernels.
- A can be any value, though it's typically between 0.3 and 0.6 (Note that a=0.5 is a 3-element kernel with the same properties.)
- Also, a *mean filter* doesn't satisfy our criteria!

Let's go to today's breakout session.

L04: Image Resizing

Today you'll be generating "image pyramids." In class, we defined the 5-element kernel for computing the Gaussian Image Pyramid as follows:

$$k = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} c \\ b \\ a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1/4 - a/2 \\ 1/4 \\ a \\ 1/4 \\ 1/4 - a/2 \end{bmatrix}$$

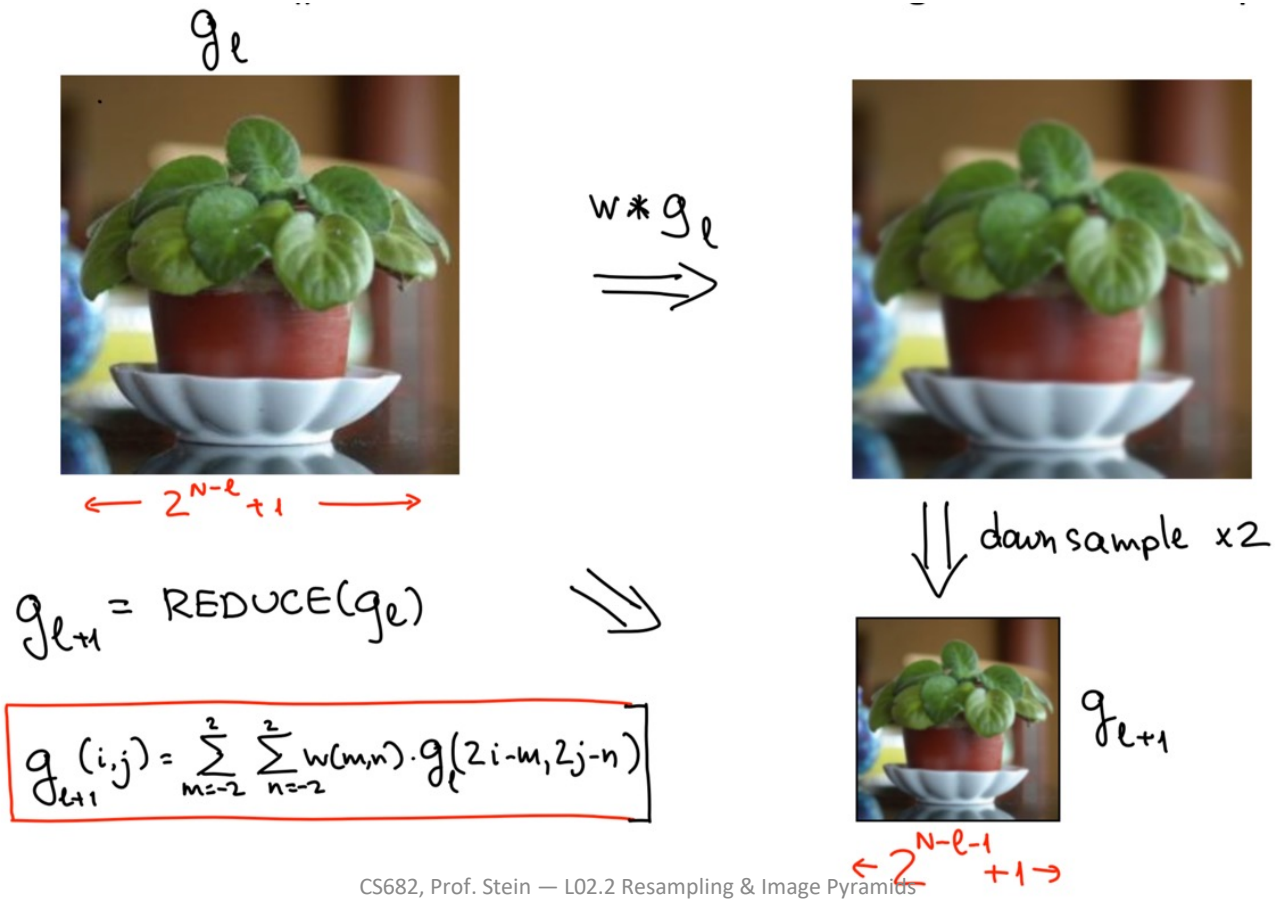
where a is a free parameter, usually between 0.3 and 0.6.

L04B.1 1D Gaussian "Image Pyramid"

you should create a Gaussian image pyramid using the kernel for different values of a. You

We can apply our new REDUCE function to travel up levels of the pyramid

For an image, we “reduce” along each axis.



Still: how wide should my filter kernel be?

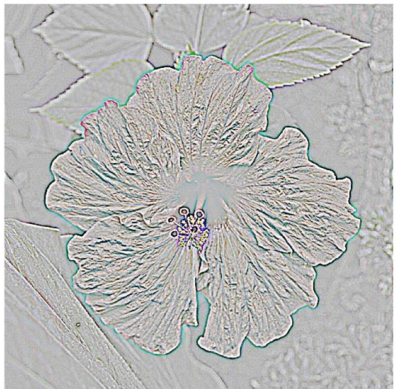
It's hard to say:

- It depends on the application (something you'll hear me say a lot this semester).
- A width of 1 is too small: we cannot achieve the Equal Contribution Criterion if the filter is smaller than 3.

So how does smoothing the image change it?

Original

Smoothed



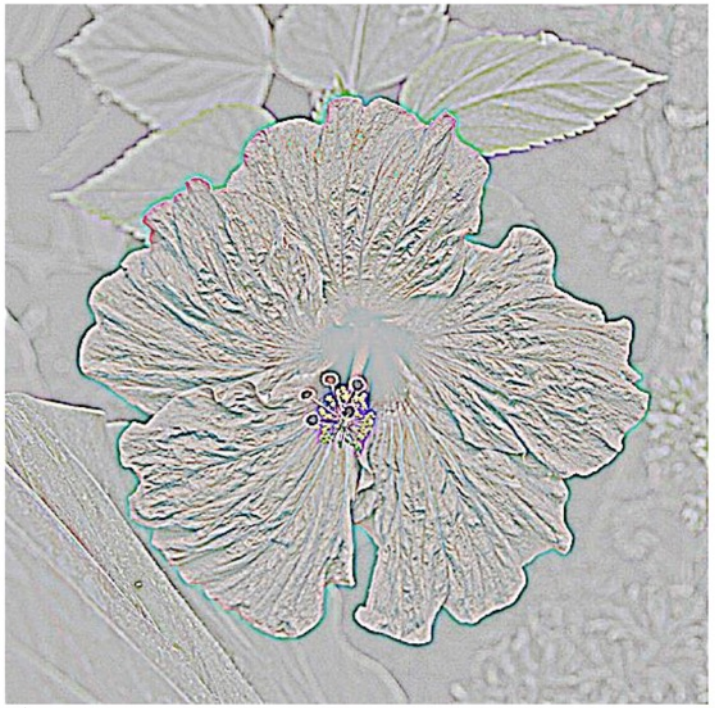
Difference
(a.k.a *residual*)

What if we take the difference between the two?

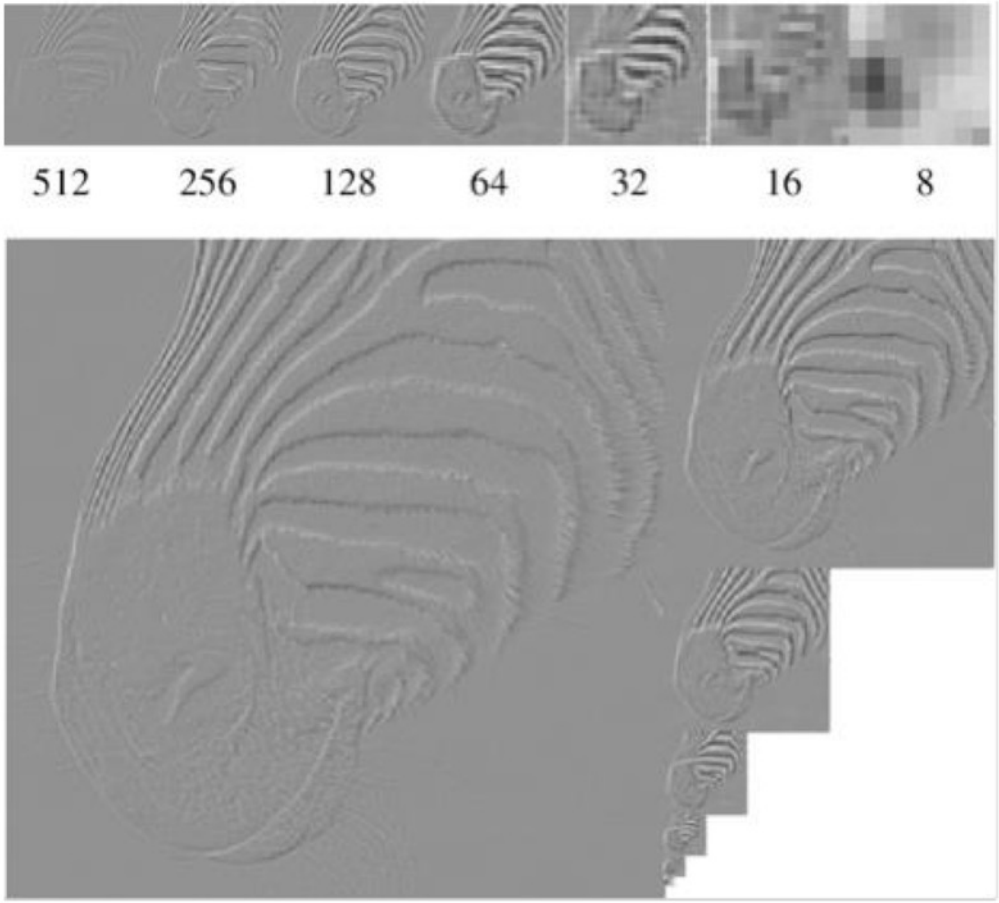
We can build a new pyramid that consists of these differences.

Gaussian vs. Laplacian Pyramids

Shown in opposite order for space.



The *Laplacian Pyramid stores the differences between different levels (instead of raw images)*

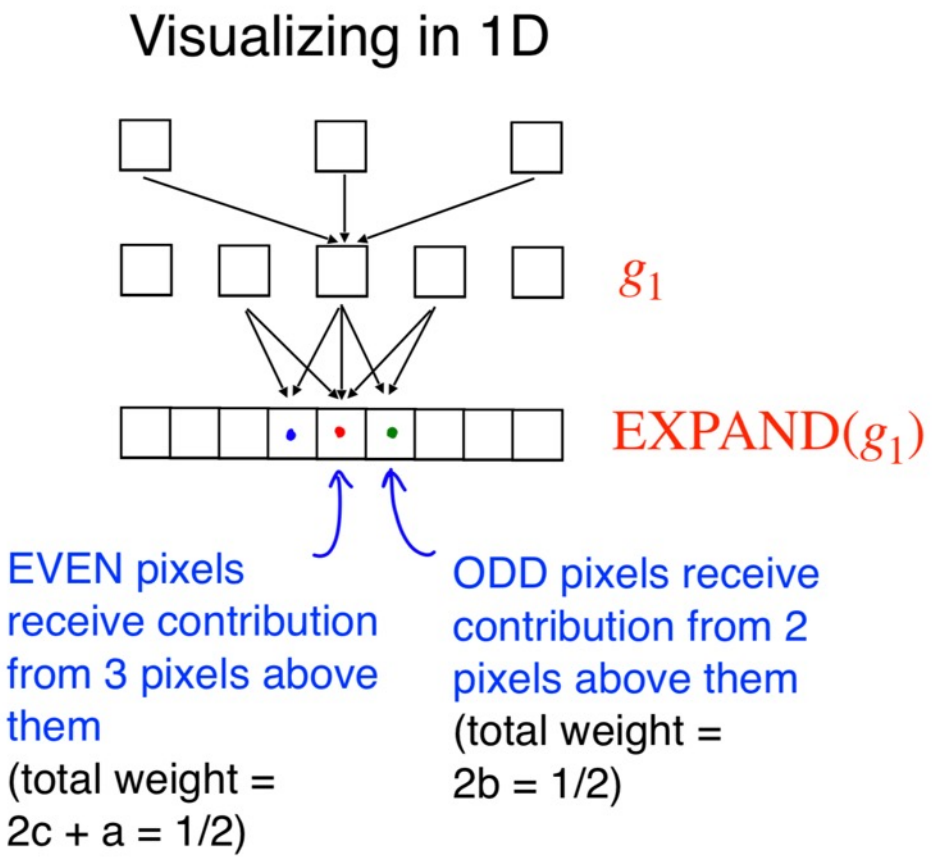


The highest level (shown as “8” on the left) stores a base image at the highest desired blur level.

Can we reconstruct the original image *perfectly* using these?

Yes! We just need an EXPAND function that is the inverse of REDUCE

The *Laplacian Pyramid stores the differences between different levels (instead of raw images)*



The highest level (shown as "8" on the left) stores a base image at the highest desired blur level.

Can we reconstruct the original image *perfectly* using these?

Yes! We just need an EXPAND function that is the inverse of REDUCE

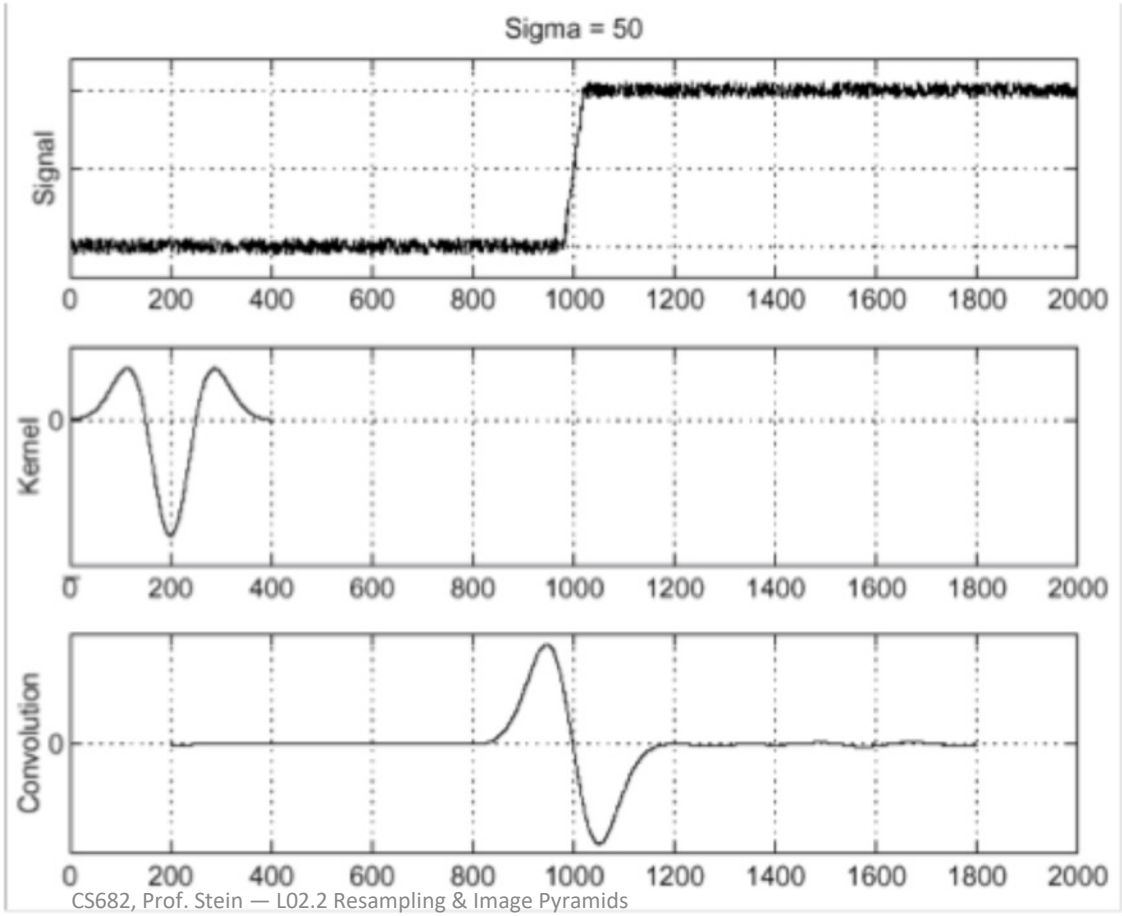
Why is it called the Laplacian Pyramid?

Remember this from last week? We can take the Laplacian of the Gaussian and use it to filter the image (which has zero crossings at edges).

input

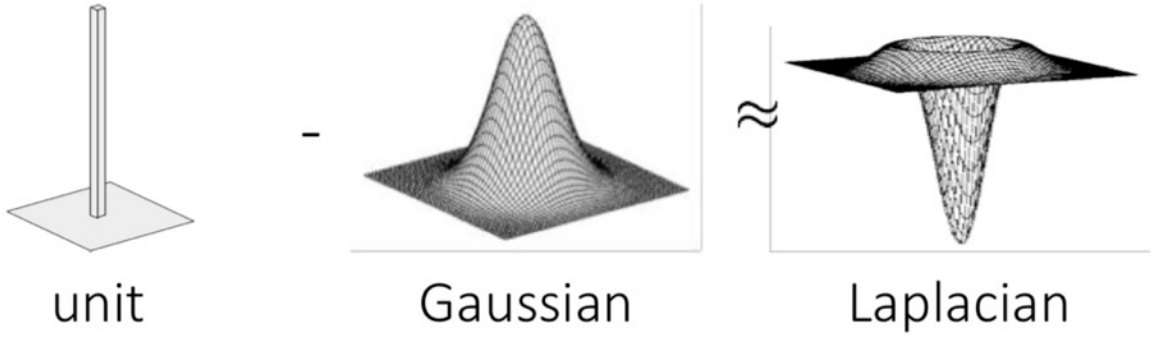
Laplacian of Gaussian

output

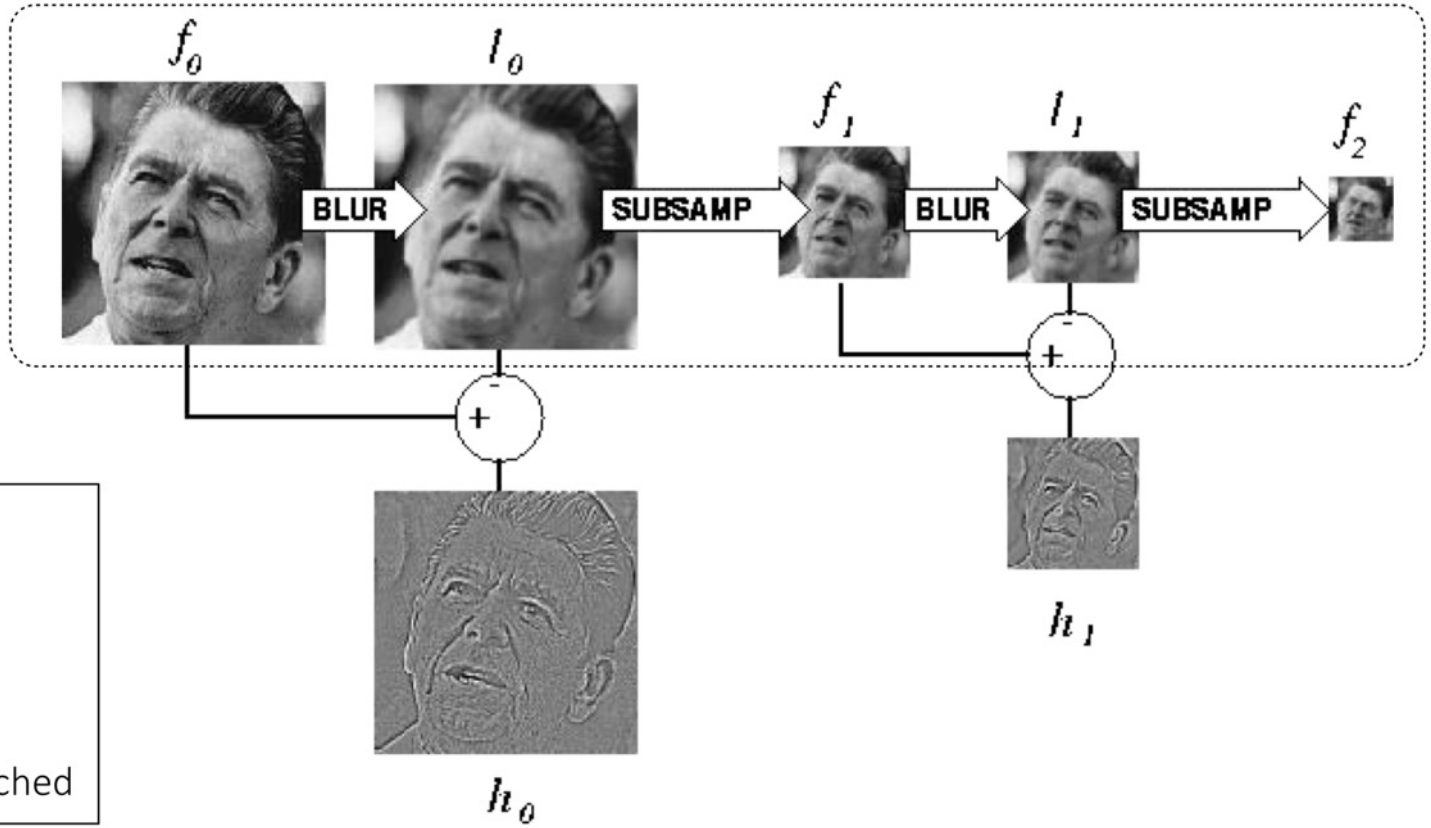


Why is it called the Laplacian Pyramid?

The difference of gaussians approximates the Laplacian.
Close enough for small kernels (and has similar properties)



Building the Laplacian Pyramid is very similar to building a Gaussian Pyramid



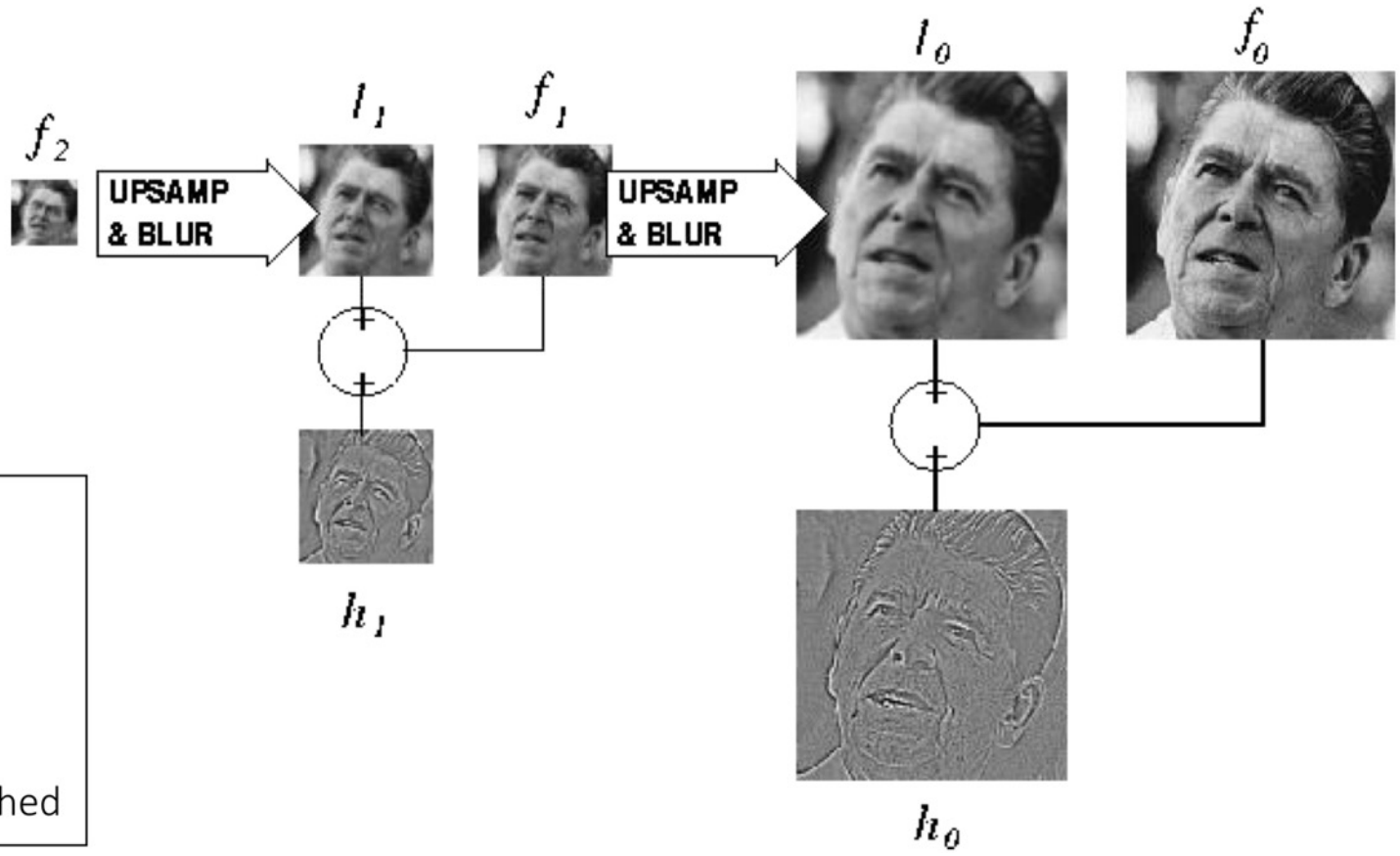
Algorithm

repeat:

- filter
- compute residual
- subsample

until min resolution reached

Reconstructing the original image requires upsampling and adding the residual



Algorithm

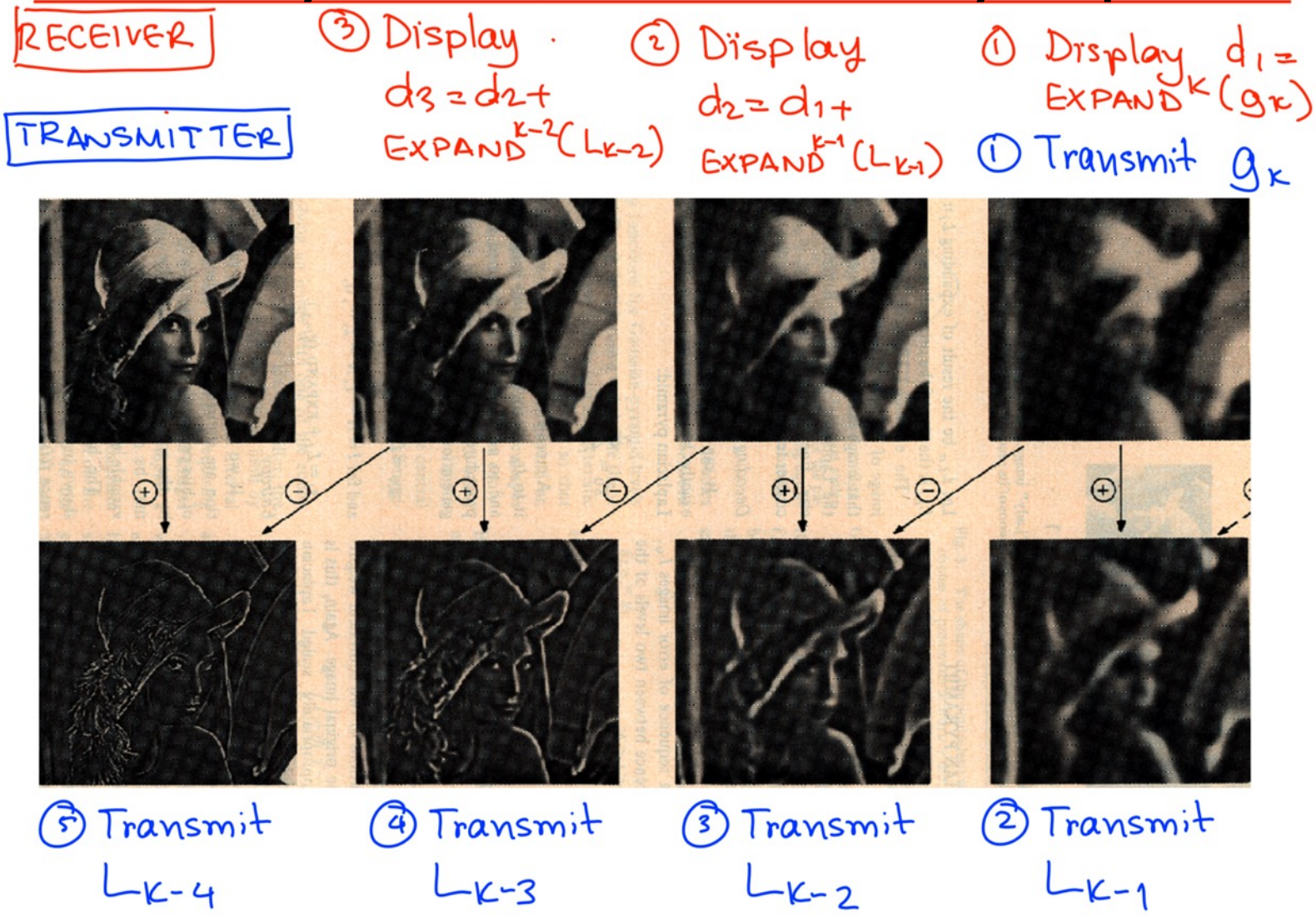
repeat:

upsample

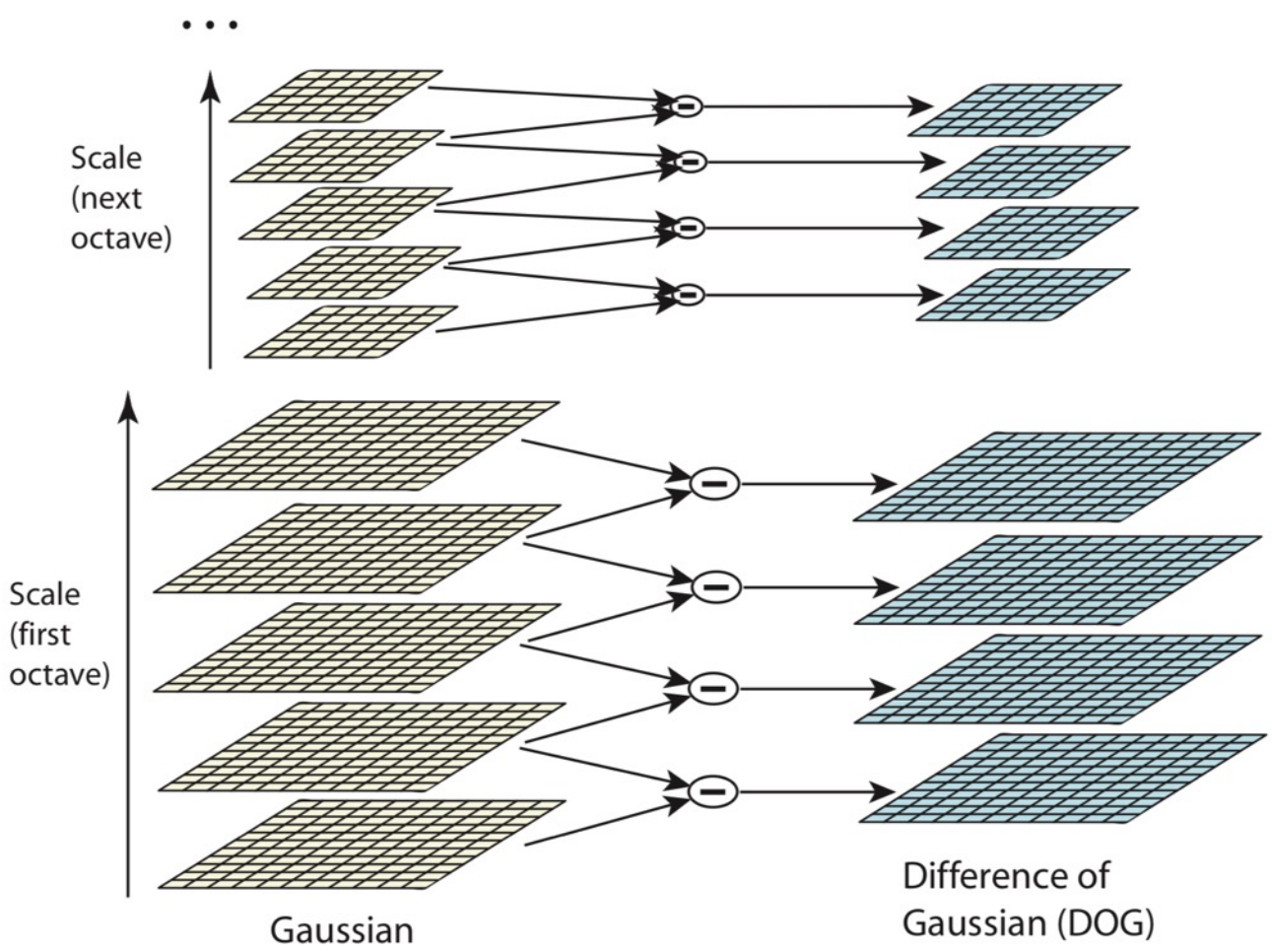
sum with residual

until orig resolution reached

Many web browsers support Progressive JPEG's: send a blurry version and iteratively improve



Aside: what if we want features at "intermediate scales"?



This is often done in feature detection (which we'll talk about in a few weeks). We blur multiple times between each downsampling operation.

*The term *scales per octave* is how many intermediate steps between downsampling by factors of 2.
See the SIFT Paper

Image Upsampling

Upsampling an Image

So, this image is small: 

How do we make it bigger?

Upsampling an Image

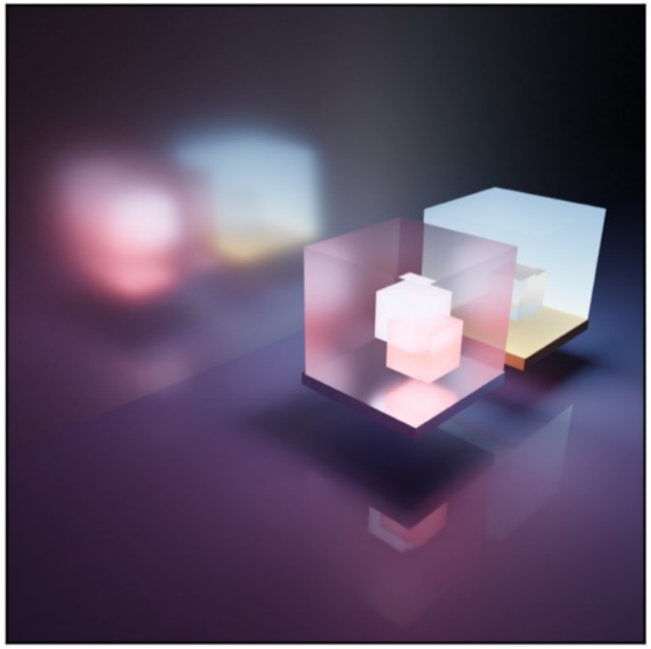
We can simply duplicate the rows and columns many times.

This is a simple form of *nearest neighbor interpolation*

Upsampling an Image

Side-by-side, we can see how *blocky* the upsampled image looks.

Orignal



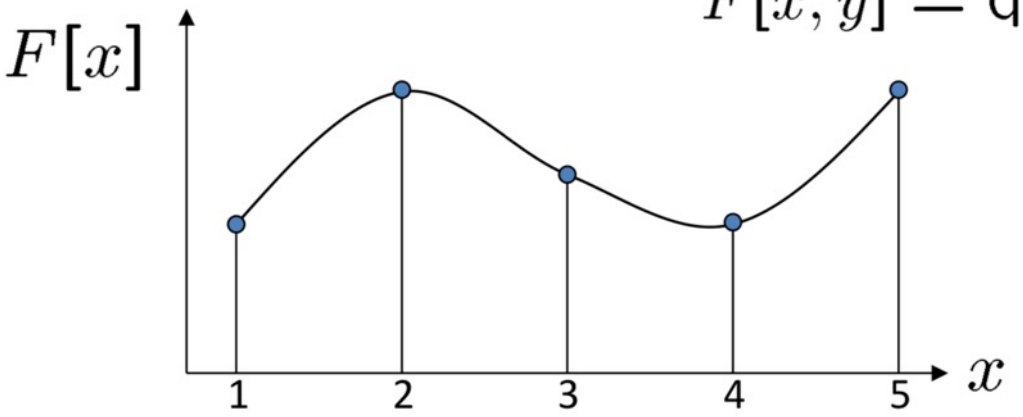
Upsamapled

How can we do better than this?

To resize an image more generally, we need to "un-discretize" it (somehow)

A digital image is a discretization of some continuous function

$$F[x, y] = \text{quantize}\{f(xd, yd)\}$$



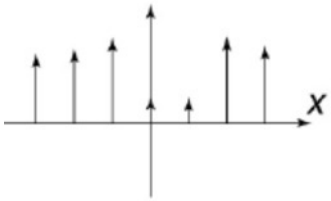
If we had access to that underlying function *f* (or could reconstruct it), we could generate images at any resolution.

So how can we reconstruct this function? *How about filtering?*

There are two ways to think about interpolation

1. We “un-discretize” an image by filling in the gaps between the values we are given.

Let’s start simple: our image is a 1D *signal*: we only know the values at these locations, but we want the values in-between:

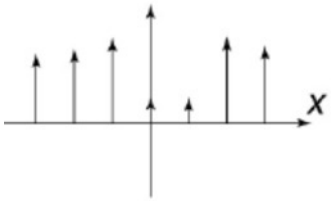


How do we do this?

There are two ways to think about interpolation

1. We “un-discretize” an image by filling in the gaps between the values we are given.

Let’s start simple: our image is a 1D *signal*: we only know the values at these locations, but we want the values in-between:

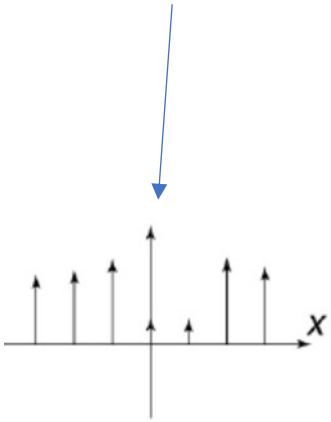


How do we do this? How about filtering!?

(It *is* the only mathematical tool we’ve developed so far…)

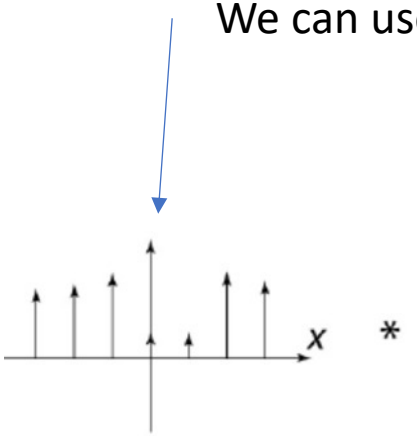
We can approximate the underlying function by convolving samples with a filter.

This is our *discrete* input.
It is just a bunch of “spikes” at each point the sample was collected.

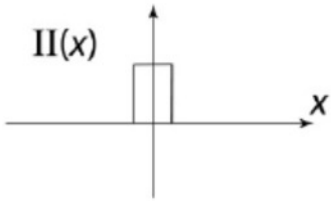


We can approximate the underlying function by convolving samples with a filter.

This is our *discrete* input.
It is just a bunch of “spikes” at each point the sample was collected.
We can use a convolution to “fatten up” those spikes:

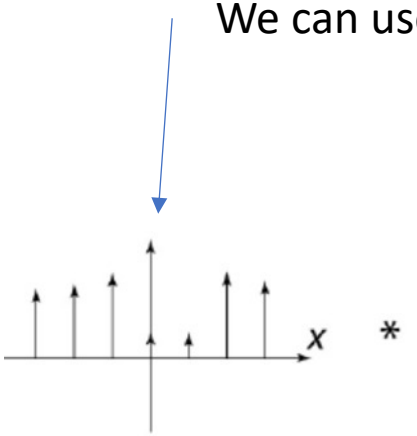


*

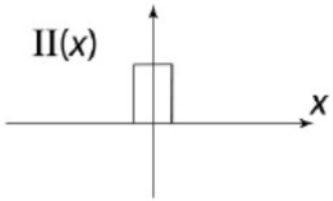


We can approximate the underlying function by convolving samples with a filter.

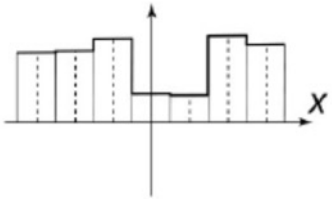
This is our *discrete* input.
It is just a bunch of “spikes” at each point the sample was collected.
We can use a convolution to “fatten up” those spikes:



*



⇒

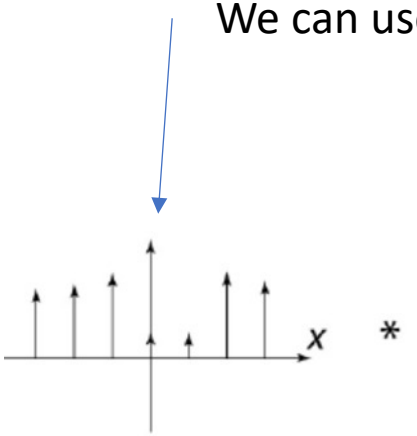


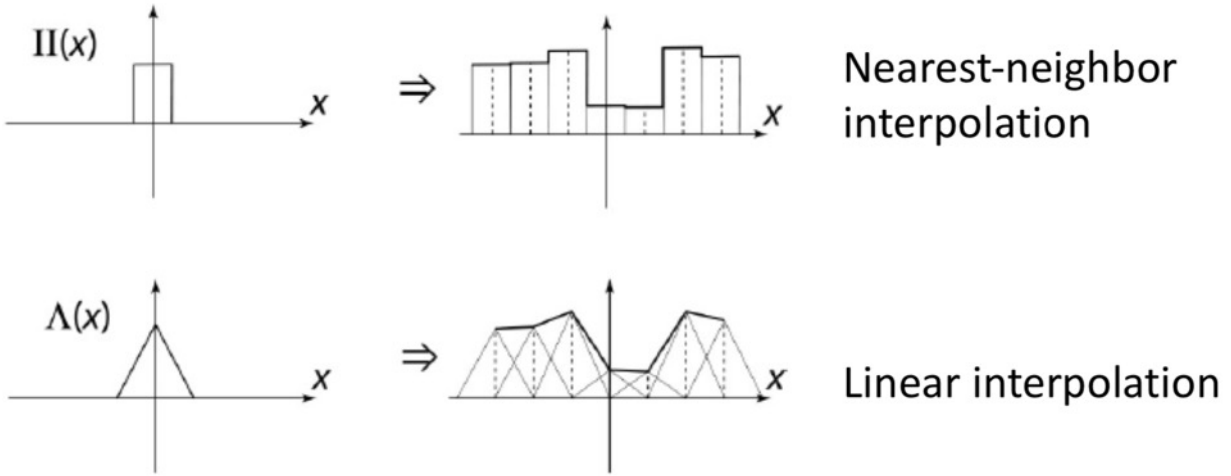
Nearest-neighbor interpolation

Each spike is “widened” to fill the gaps between each.

There are many choices of filter function for image interpolation

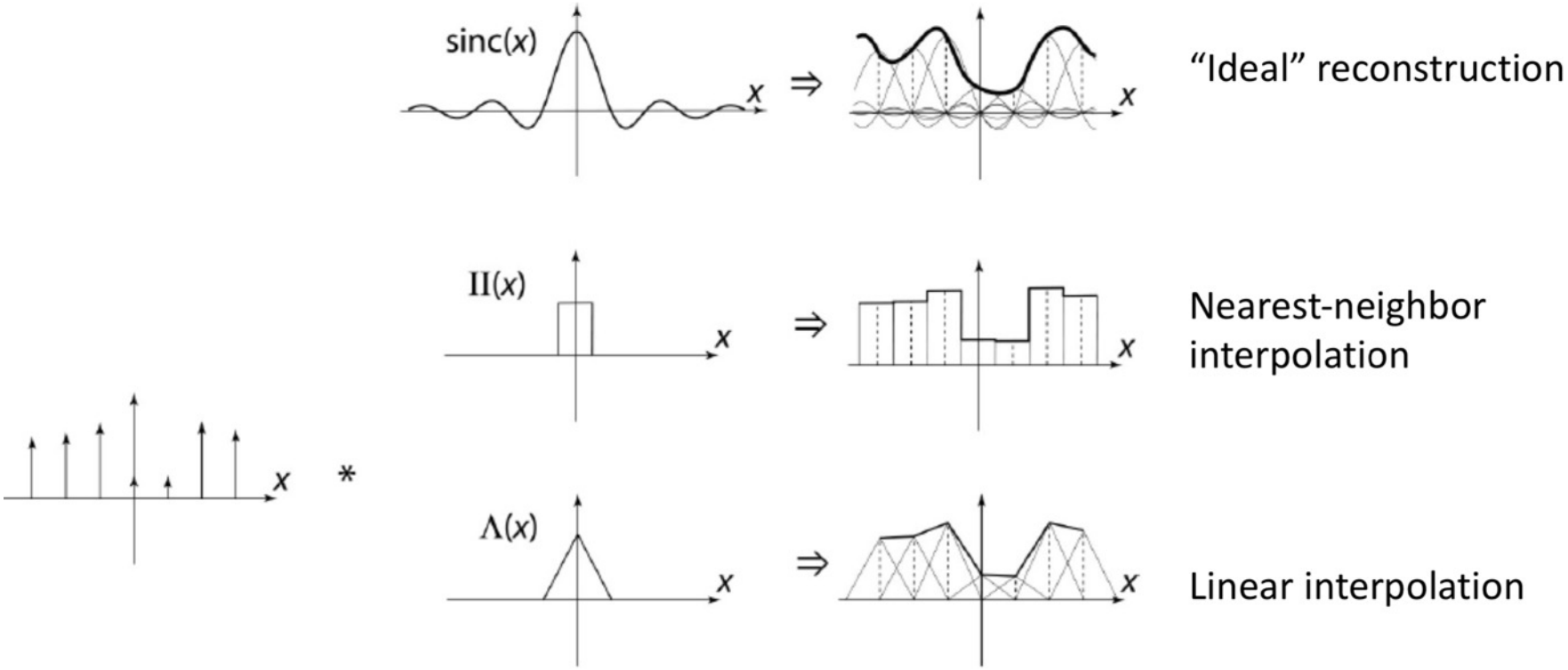
This is our *discrete* input.
It is just a bunch of “spikes” at each point the sample was collected.
We can use a convolution to “fatten up” those spikes:



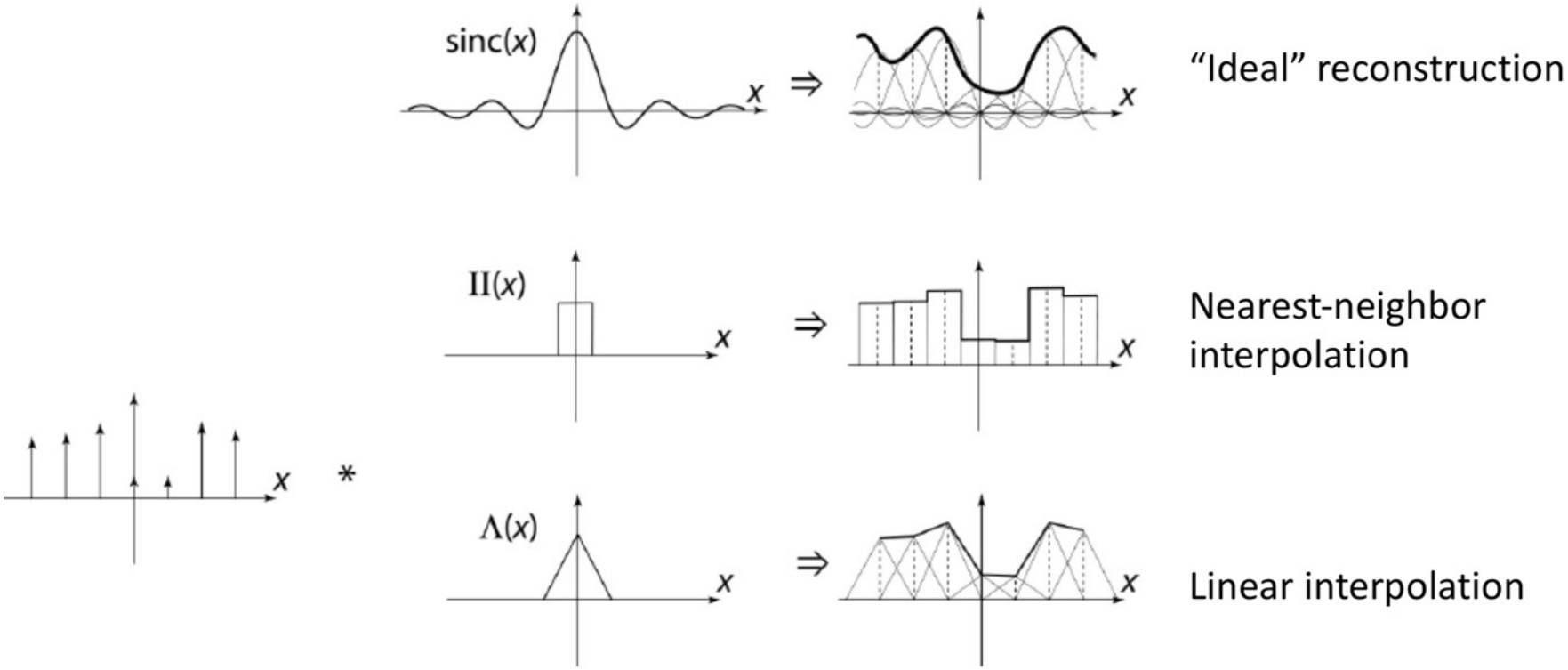


[Let’s go to the board to visualize this one.]

There are many choices of filter function for image interpolation



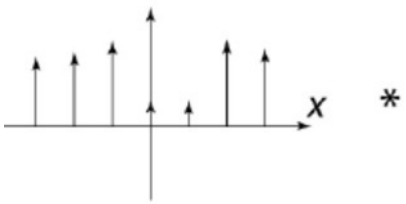
There are many choices of filter function for image interpolation

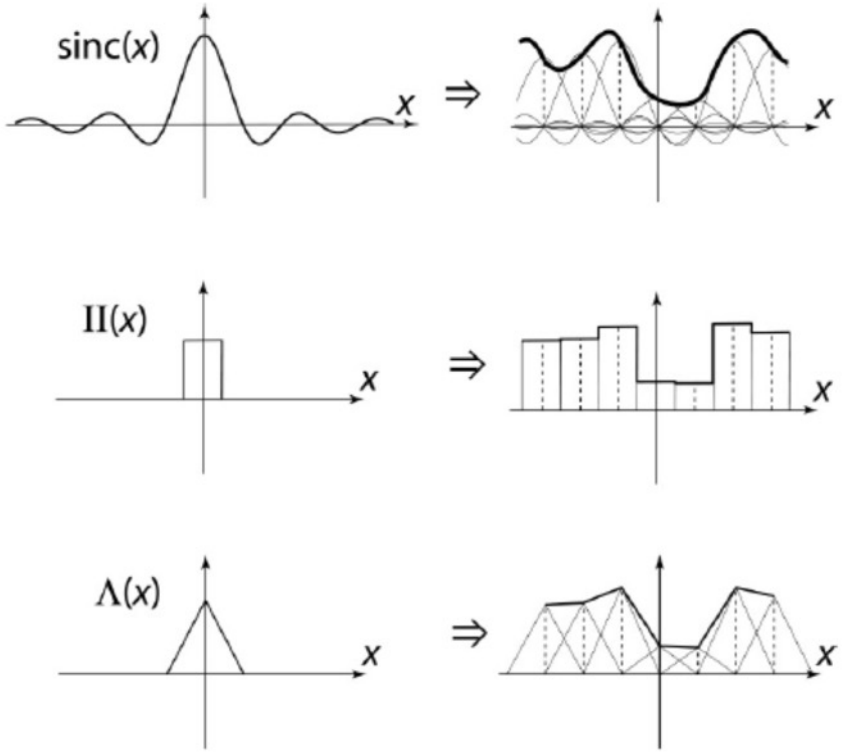


More complex functions (like a cubic curve) can also be used.
You'll explore this more in your programming assignment!

There are many choices of filter function for image interpolation

Why might this function be considered "ideal"? Think of the Fourier Transform.





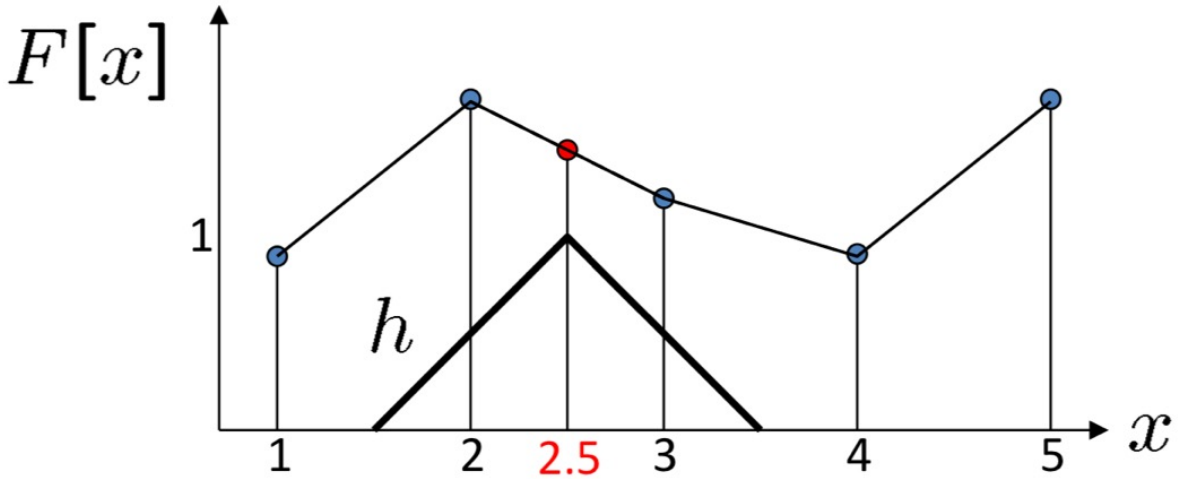
More complex functions (like a cubic curve) can also be used.
You'll explore this more in your programming assignment!

There are two ways to think about interpolation

1. We “un-discretize” an image by filling in the gaps between the values we are given.

There are two ways to think about interpolation

1. We "un-discretize" an image by filling in the gaps between the values we are given.
2. We can use the same filter kernel to compute a weighted sum of nearby pixels:



Instead of convolving each pixel (the inputs), we use the filter at each "output location" to compute the contribution of each input. *These are mathematically equivalent. You'll be using #2 in your assignment.*

For now, we can think of 2D interpolation as 2-1D interpolations

We can interpolate first along one axis, and then along the other.

Alternatively, you can use a single 2D kernel to represent the 2D operation with a “single application.”

In practice, there are plenty of ways to implement the same operation.

For now, we can think of 2D interpolation as 2-1D interpolations

We can interpolate first along one axis, and then along the other.

Alternatively, you can use a single 2D kernel to represent the 2D operation with a “single application.”

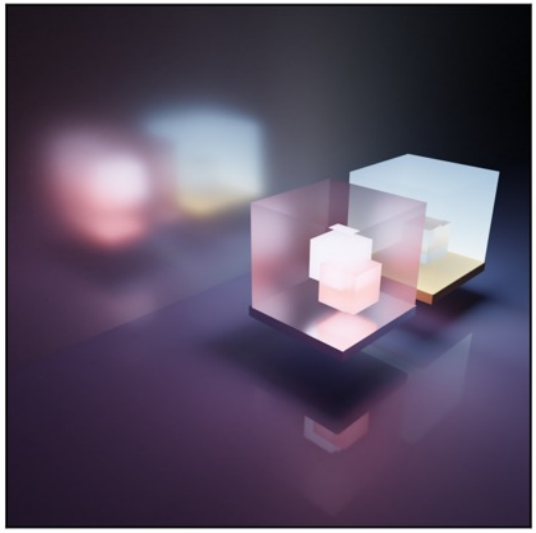
In practice, there are plenty of ways to implement the same operation.

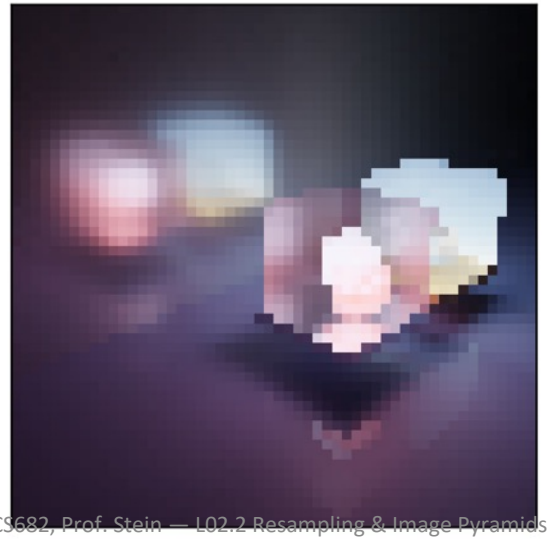
Note: for image warping (a generalization of this idea), when we rotate and stretch and distort the image, we will not be able to easily “handle” each dimension separately.

Upsampling can be done using Fourier Transforms if you'd like.

Procedure:

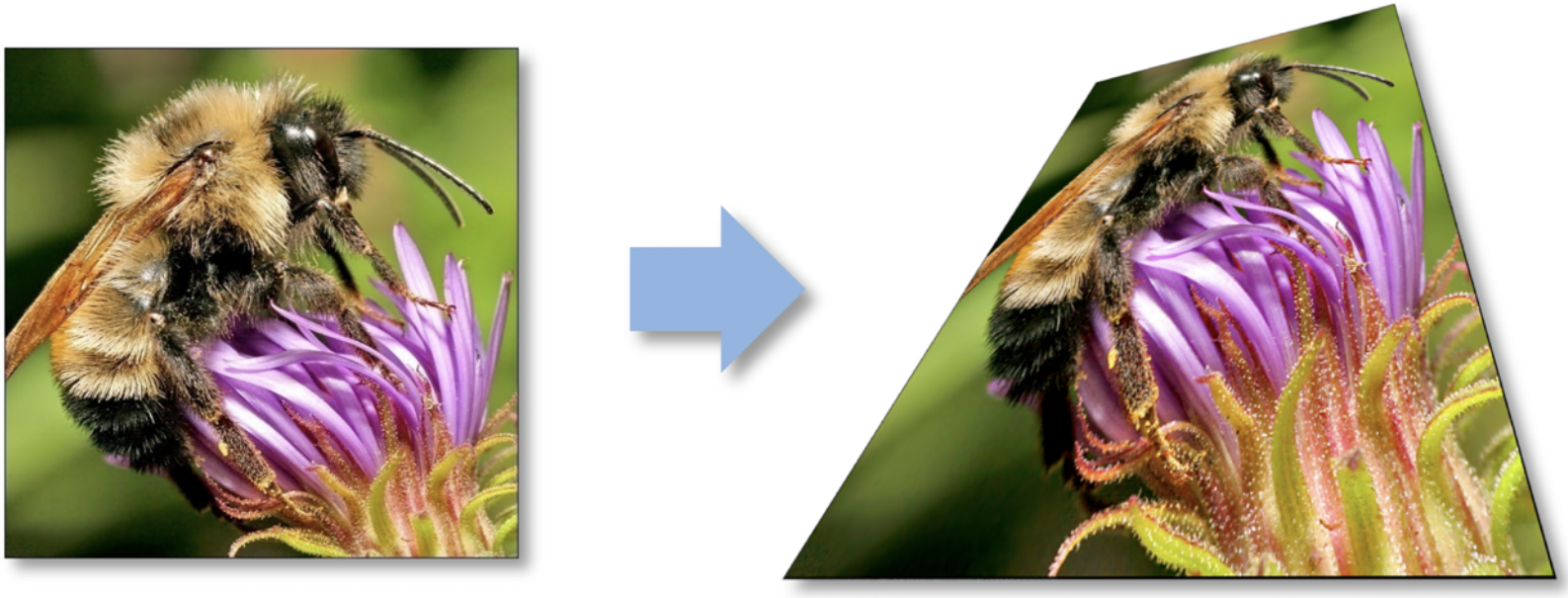
1. Convert the image to frequency space
2. "Pad" the frequency-space image with zeros up to the desired size (be careful about normalization and the centering of the DFT frequency vector)
3. Transform back





Interpolation is also used during *image warping*.

When transforming an image, we often need to find the value of the image function between pixels and then re-discretize:



We will talk about warping again in a couple of weeks.