Domain Transforms, Warping & Morphing

Fredo Durand
MIT EECS
6.8370/6.8371

Domain operations

Domain transform

- Apply a function *f* from R^2 to R^2 to the image domain
- if *(x, y)* had color *c* in the input,
  f(x, y) has color *c* in the output

Transformation

- **Simple parametric transformations**
  - linear, affine, perspective, etc



translatio
n



rotation



aspect



affine



perspective



cylindrical

illustration by Rick Szeliski

Warping

- Imagine your image is made of rubber
- warp the rubber



No prairie dogs were armed when creating this image

Application of warping: weight loss

- **Liquify in photoshop**





figure 9.37

Selecting the entire left side of the image avoids potential artifacts.



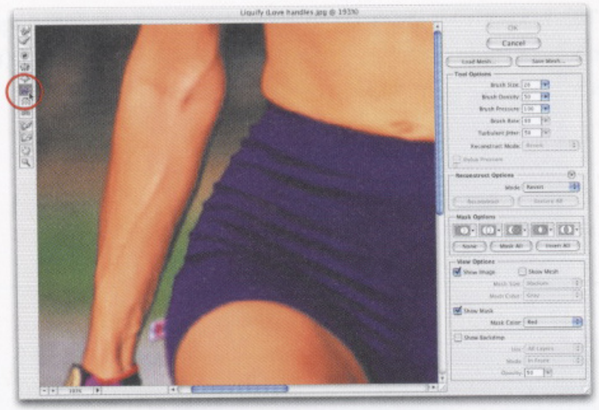
figure 9.38

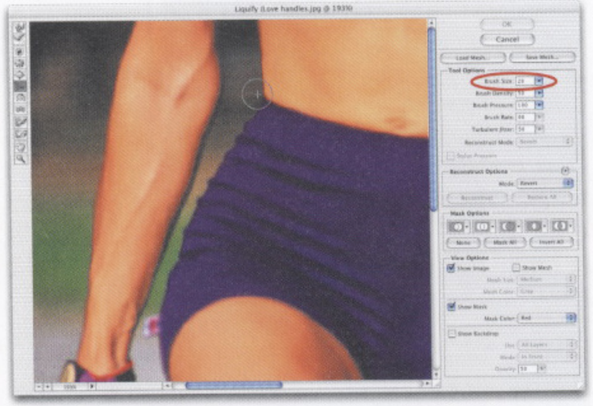
Dragging a Free Transform handle to narrow the selected area.



figure 9.39

The Liquify filter's Warp tool pushes pixels forward as you drag.



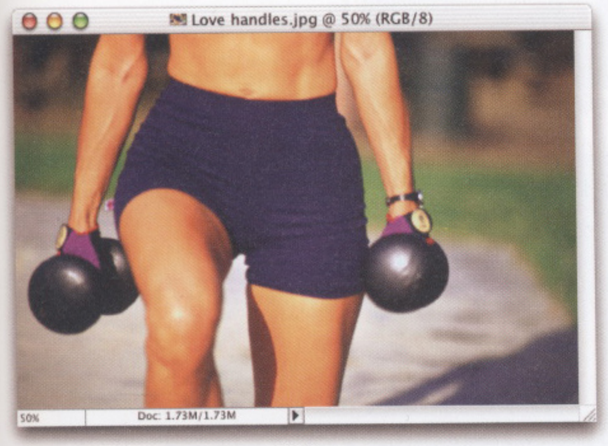


Step Three:

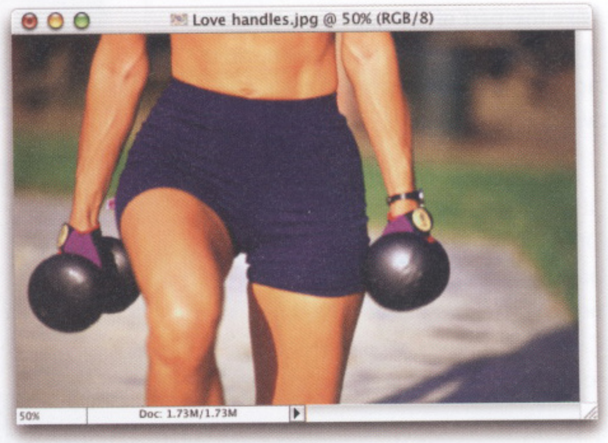
Get the Push Left tool from the Toolbar (as shown here). It was called the Shift Pixels tool in Photoshop 6 and 7, but Adobe realized that you were getting used to the name, so they changed it, just to keep you off balance.

Step Four:

Choose a relatively small brush size (like the one shown here) using the Brush Size field near the top-right of the Liquify dialog. With it, paint a downward stroke starting just above and outside the love handle and continuing downward. The pixels shifts back in toward the body, removing the love handle as you paint. (Note: If you need to remove love handles on the left side of the body, paint upward rather than downward. Why? That's just the way it works.) When you click OK, the love handle repair is complete.



Before.



After.

Domain transform issues

- Apply a function *f* from R2 to R2 to the image domain
- looks easy enough

- But 2.5 big issues:
  - which direction do we transform
  - how do we deal with non-integer coordinates?

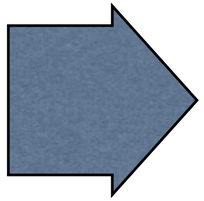
  - And for warping: how do we specify *f*?

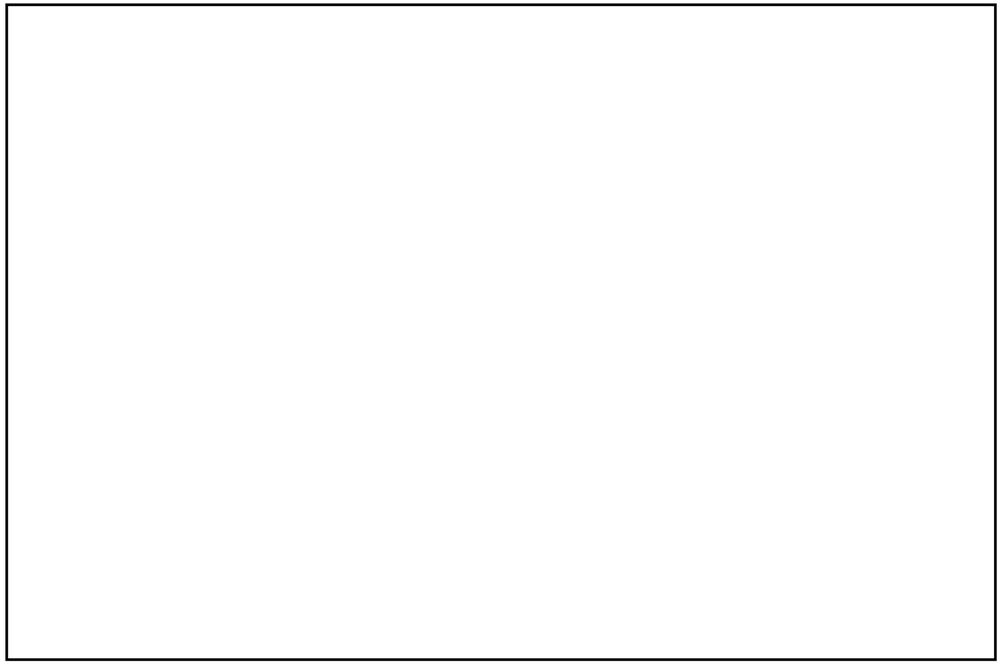
Questions?

Basic resampling

Naive scaling





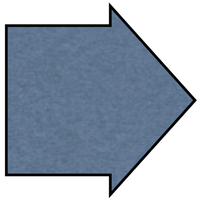


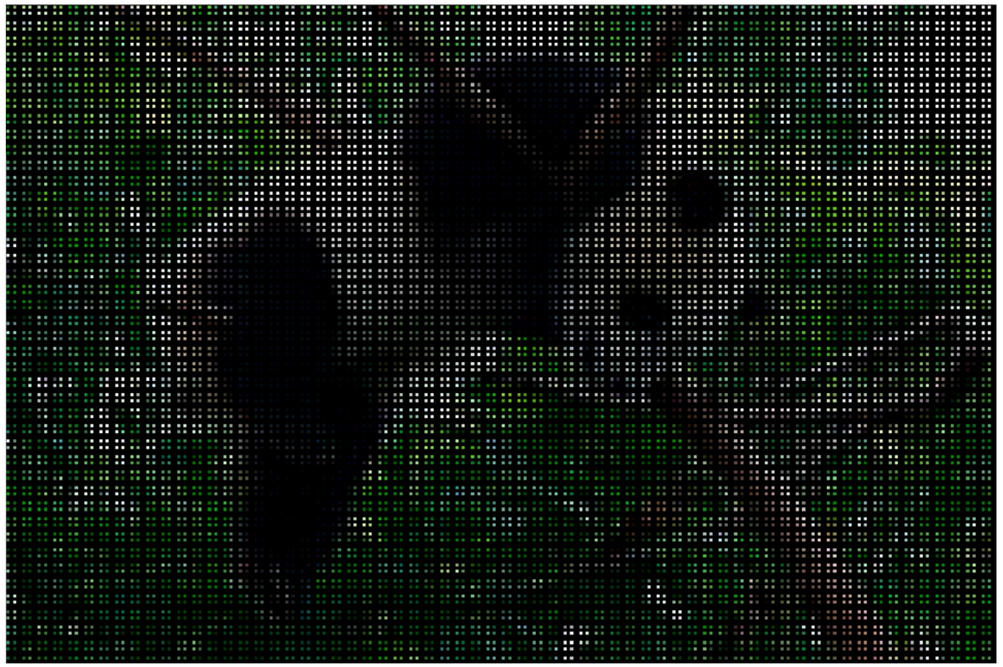
Naive scaling

- Take all the pixels in input and transform them to their outp location
  - im[y, x] => out[k*y, k*x]

```
def scaleBAD(im, k):
    out = constantIm(im.shape[0]*k, im.shape[1]*k, 0)
    for y, x in imIter(im):
        out[k*y, k*x]= im[y, x]
    return out
```





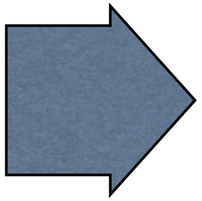


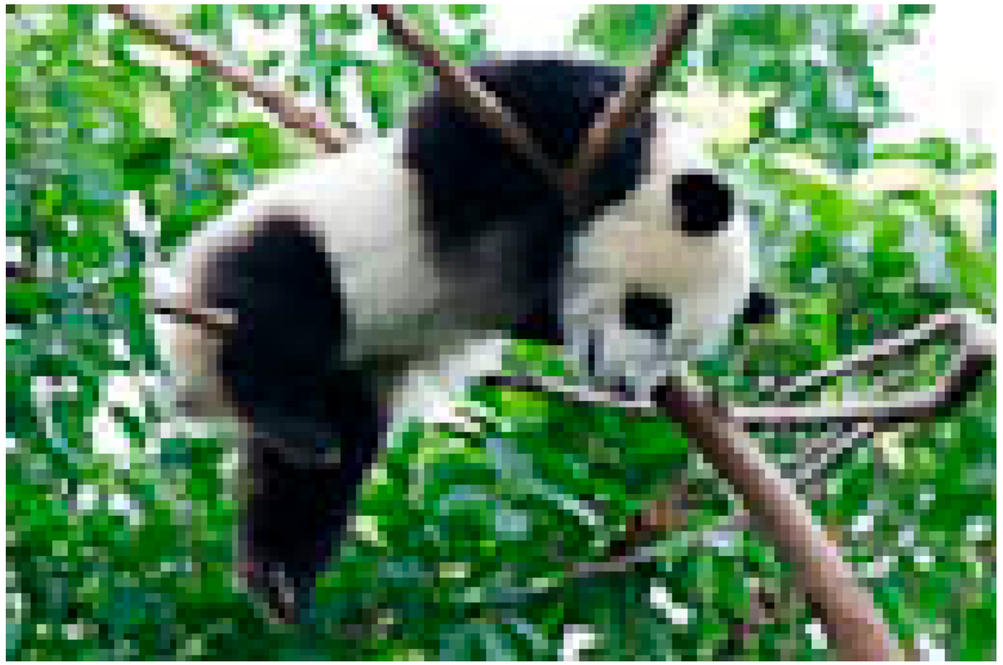
Use the inverse transform!!!!!

- Main loop on **output** pixels
  - out[y, x] <=im[y/k, x/k]

```
def scaleWithInverse(im, k):
    out = constantIm(im.shape[0]*k, im.shape[1]*k, 0.0)
    for y, x in imIter(out):
        out[y,x]=im[y/k, x/k]
    return out
```







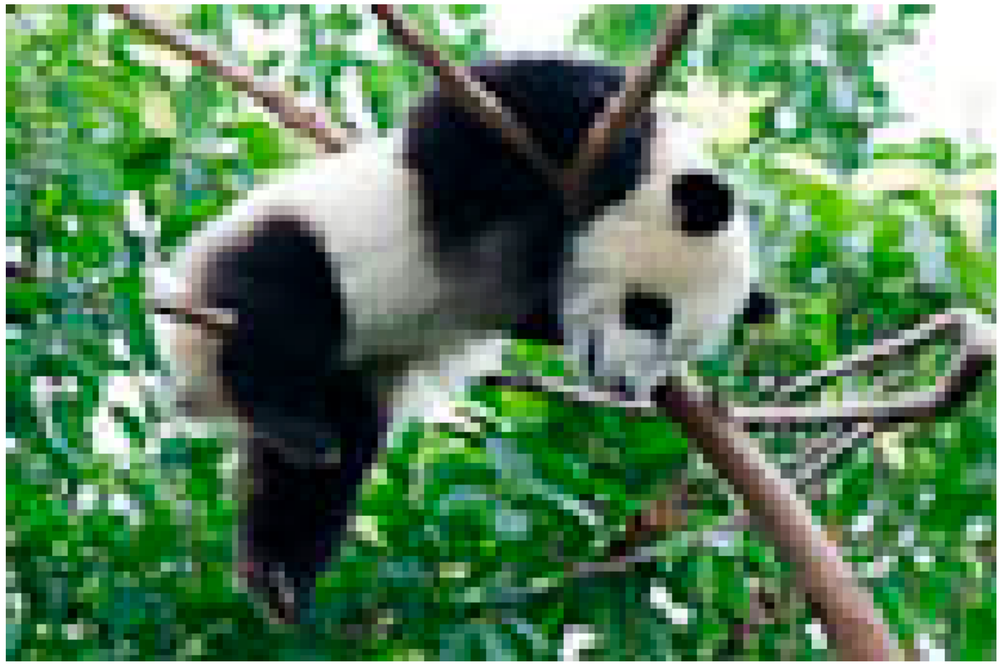
Take-home message

- Main loop over OUTPUT pixels
- use INVERSE transform

- Questions?

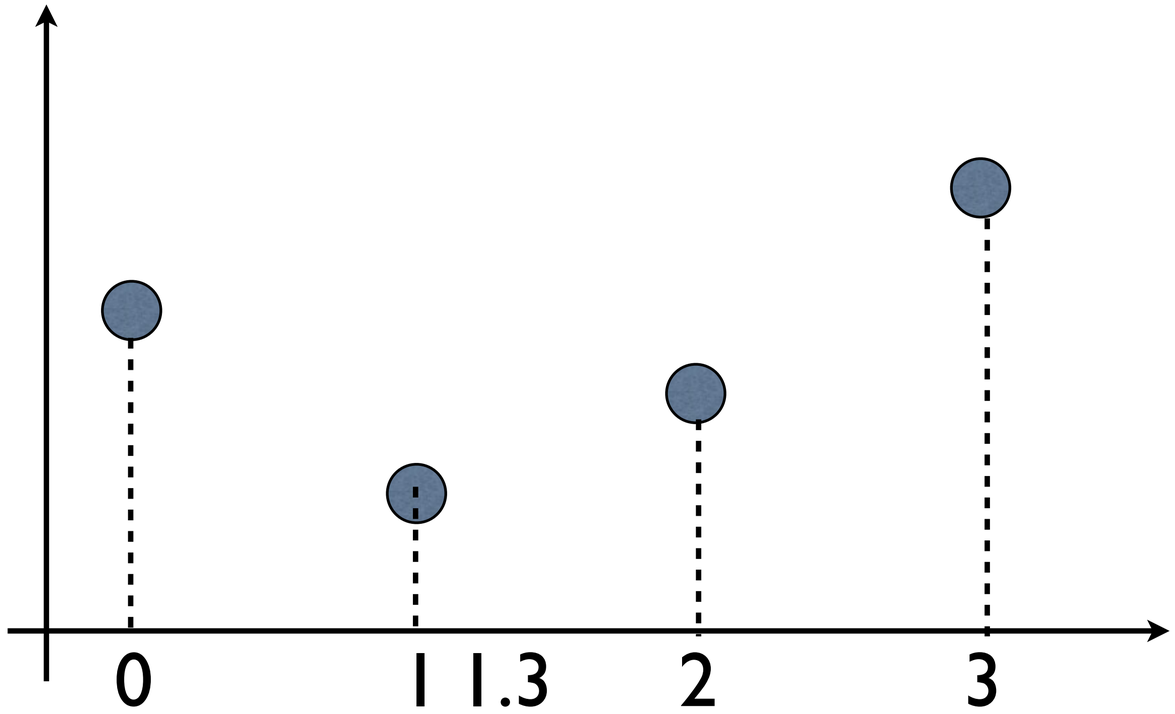
Remaining problem

- A little too “blocky”
- Because we round to the nearest integer pixel coordinates
  - called nearest neighbor reconstruction



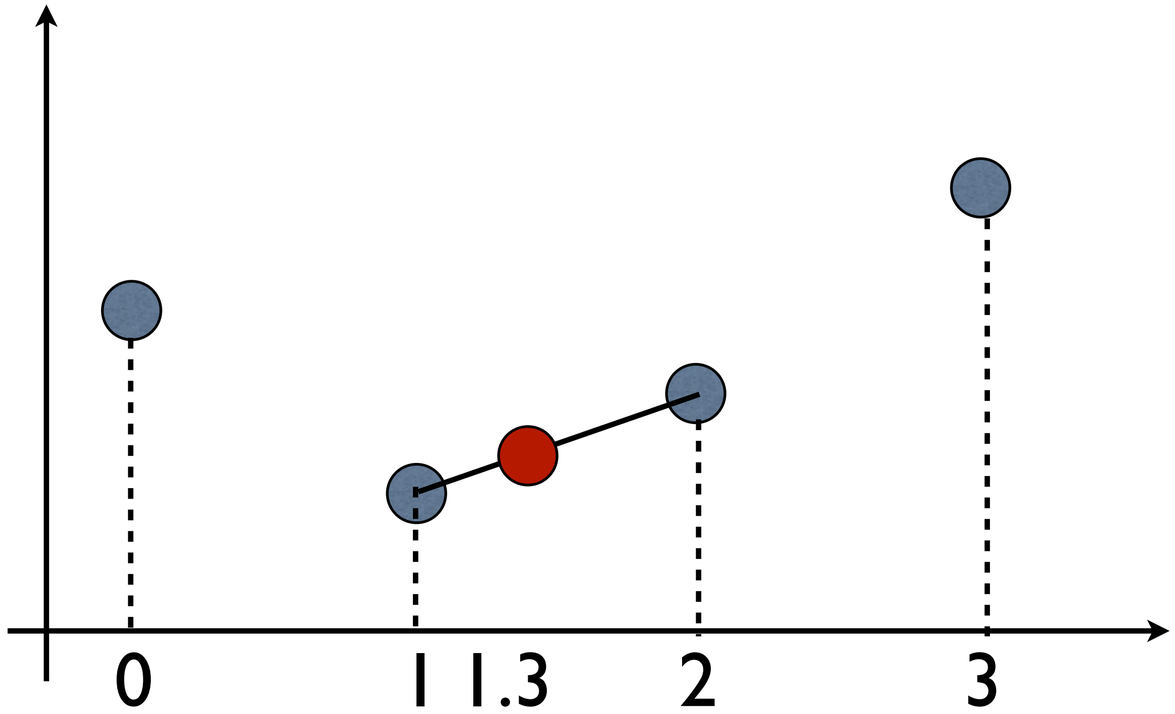
Linear reconstruction - 1D

- im is now a 1D array along x
- reconstruct im[1.3]



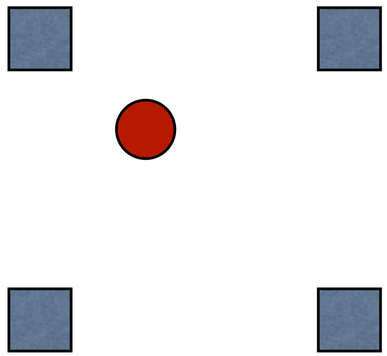
Linear reconstruction - 1D

- im is now a 1D array along x
- reconstruct im[1.3]
- =0.7*im[1]+0.3*im[2]

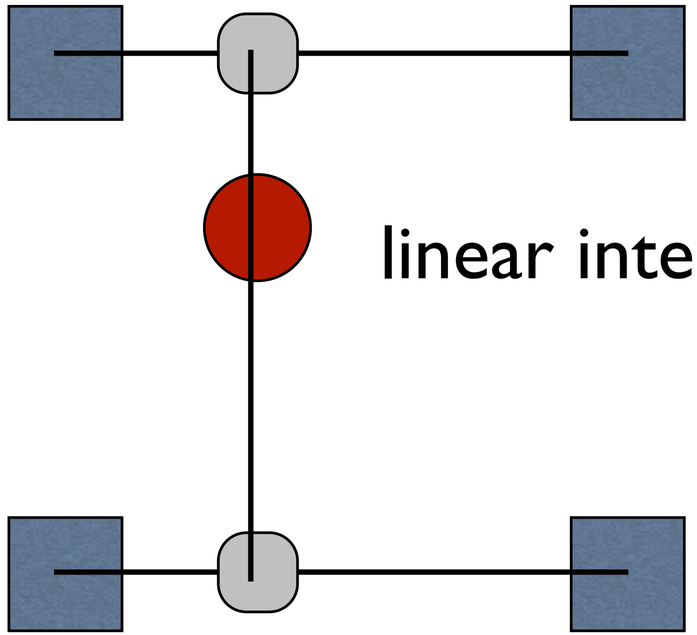


Bilinear reconstruction

- Take 4 nearest neighbors
- Weight according to x & y fractional coordinates
- Can be done using two 1D linear reconstructions along x then y (or y then x)



Bilinear



linear interpolation along x

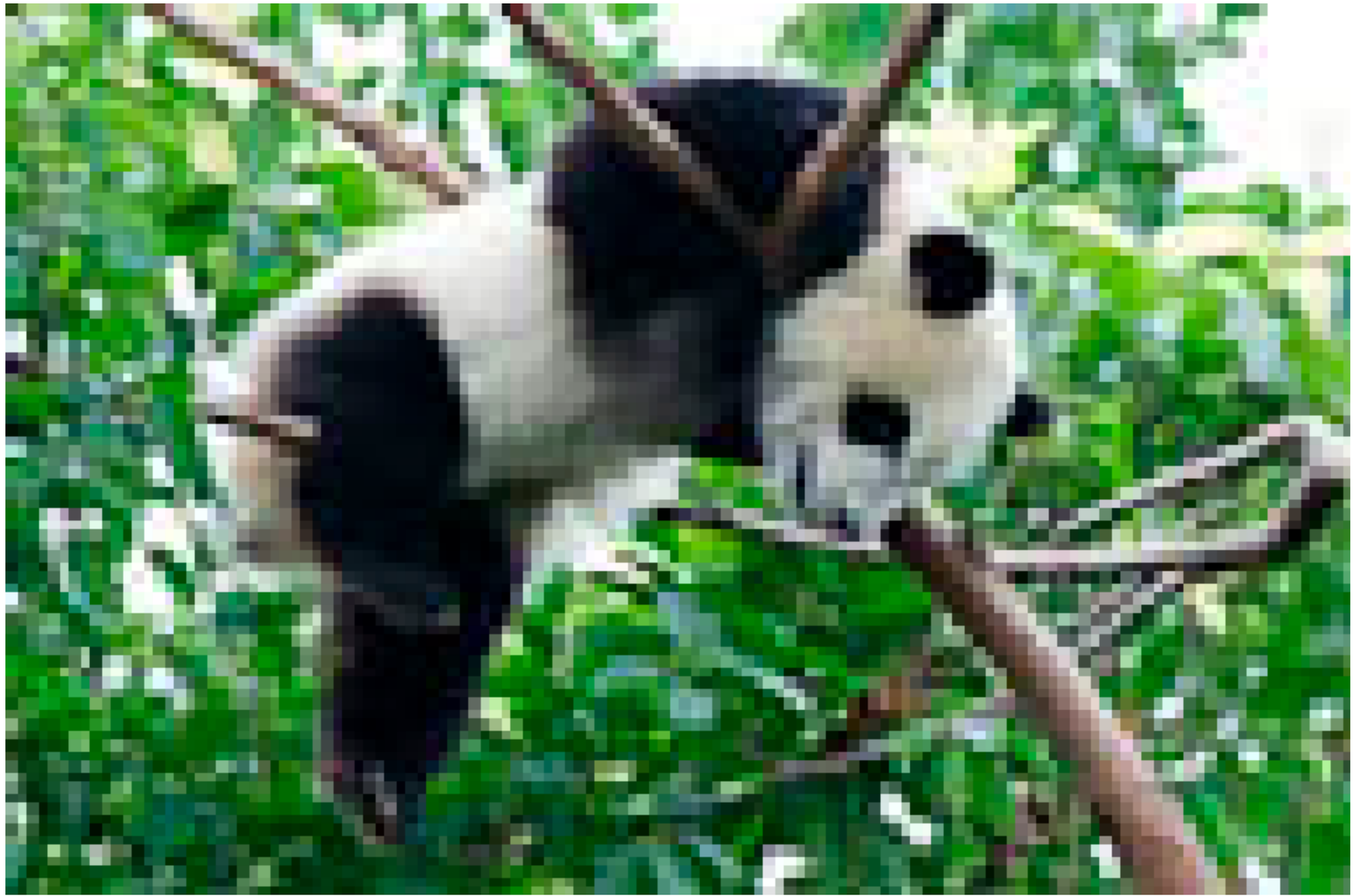
linear interpolation along y

linear interpolation along x

Exercise

- Prove it’s the same if you interpolate along y first

Recall nearest neighbor

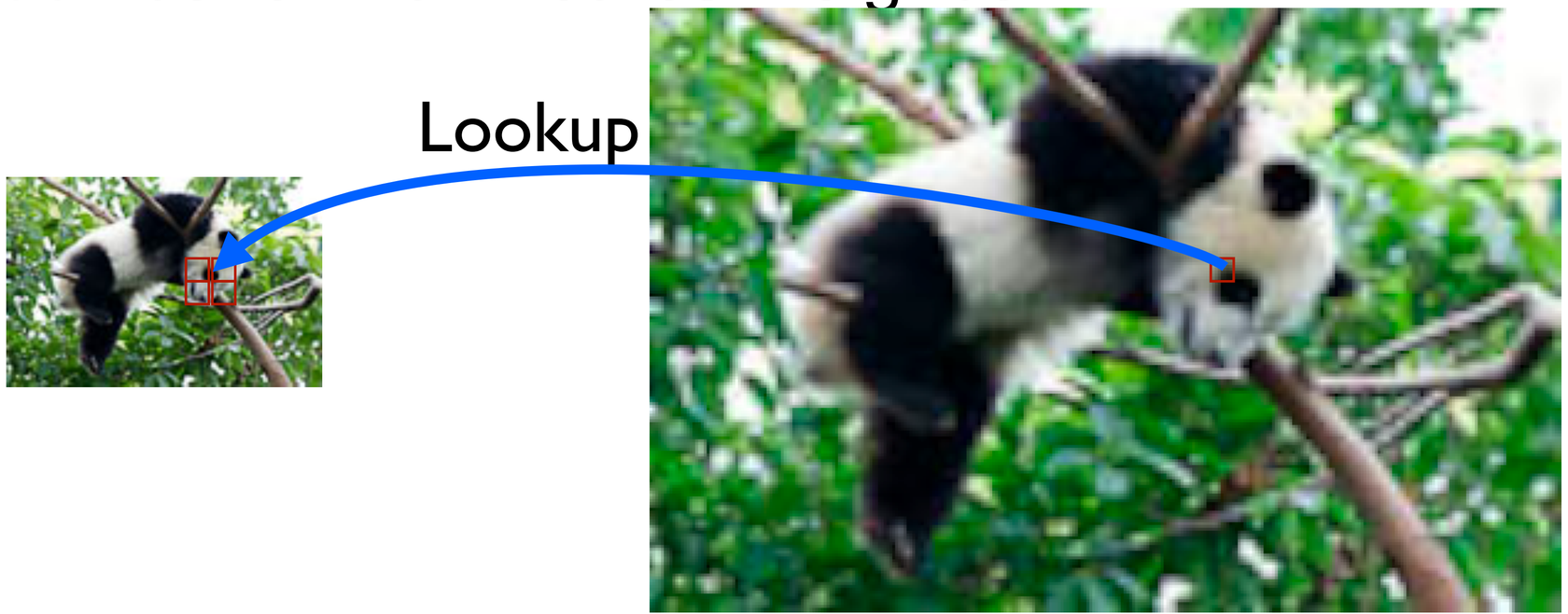


Bilinear



Take home messages

- **Main loop over OUTPUT pixels**
  - Makes sure you cover all of them
- **Use inverse transform**
- **Reconstruction makes a difference**
  - Linear much better than nearest neighbor



Questions?

General & Bicubic Resampling

Idea

- Output is weighted average of neighbors in input
- Weights given by ANALYTICAL convolution kernel
- centered at non-integer coordinates (desired output location)

Idea in 1D

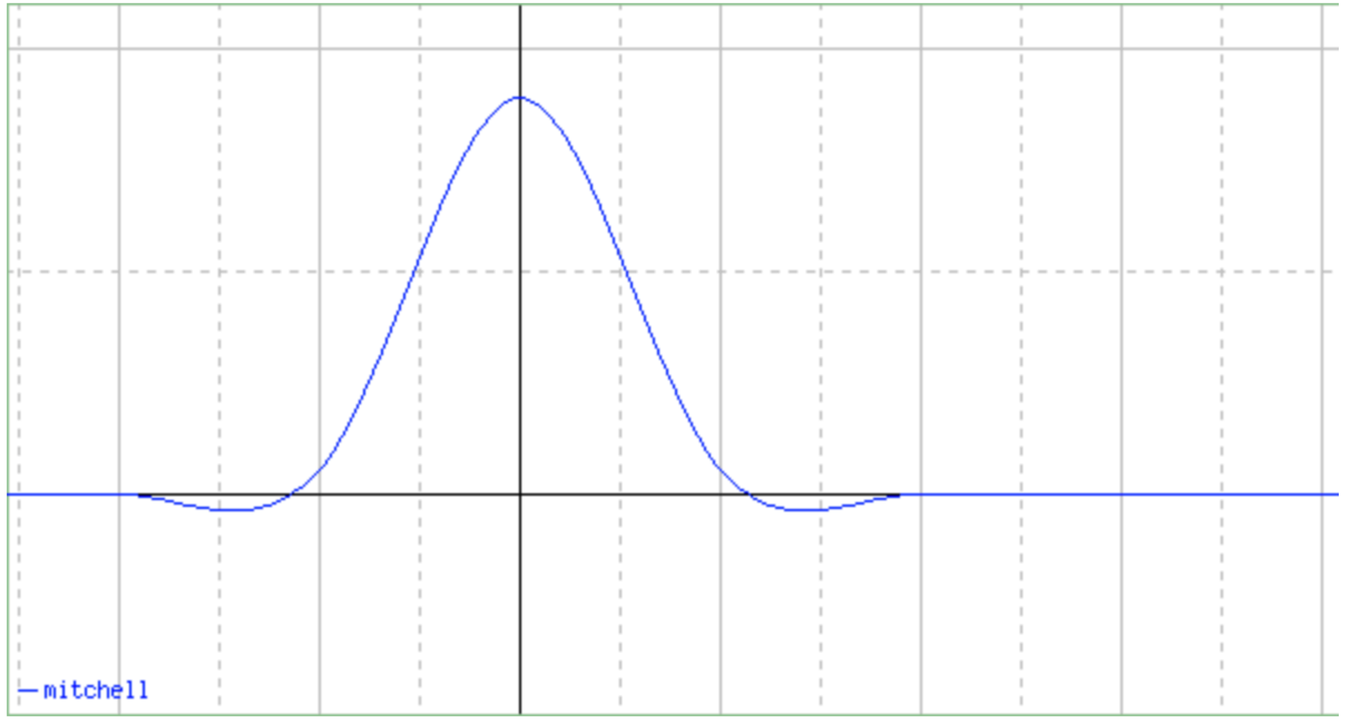
Reinterpreting bilinear

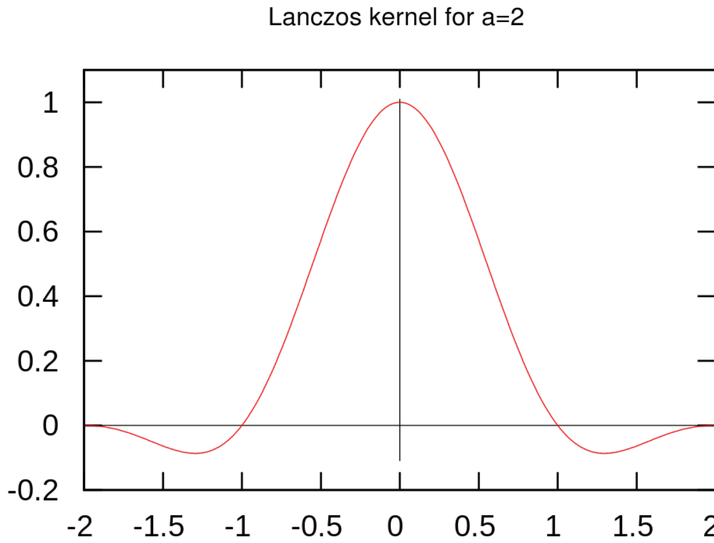
- (Because earlier exposition does NOT generalize to bicubic)

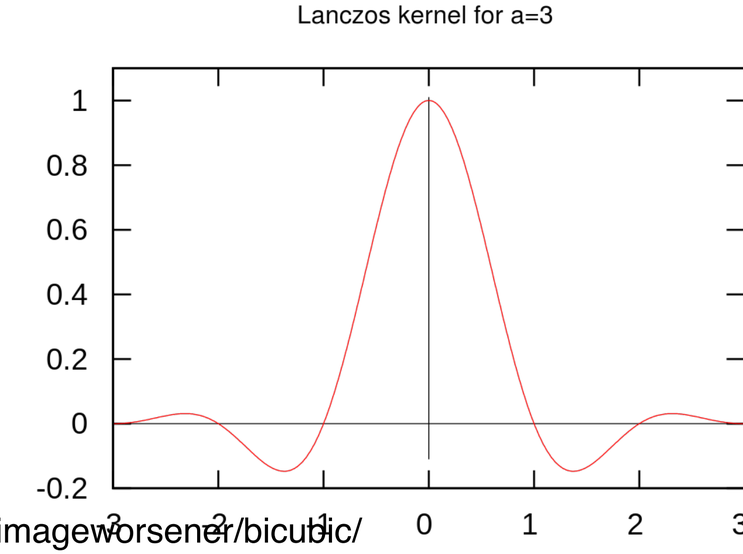
How do we express the kernel?

Some kernels

- Lanczos: sinc(π x) sinc(π x/a)
- Various bicubic







http://entropymine.com/imageworsener/bicubic/

What this is NOT

- It is NOT about fitting a cubic to the data
- Don’t believe what you see online or even in books

Mitchell/Netravali bicubic filterS

- https://www.cs.utexas.edu/~fussell/courses/cs384g-fall2013/lectures/mitchell/Mitchell.pdf

Reconstruction Filters in Computer Graphics

Don P. Mitchell
Arun N. Netravali

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

$$k(x) = \frac{1}{6}\begin{cases}(12-9B-6C)|x|^3+ & \text{if } |x|<1\\ \quad(-18+12B+6C)|x|^2+(6-2B) & \\ (-B-6C)|x|^3+(6B+30C)|x|^2+ & \text{if } 1\le|x|<2\\ \quad(-12B-48C)|x|+(8B+24C) & \\ 0 & \textit{otherwise}\end{cases}$$

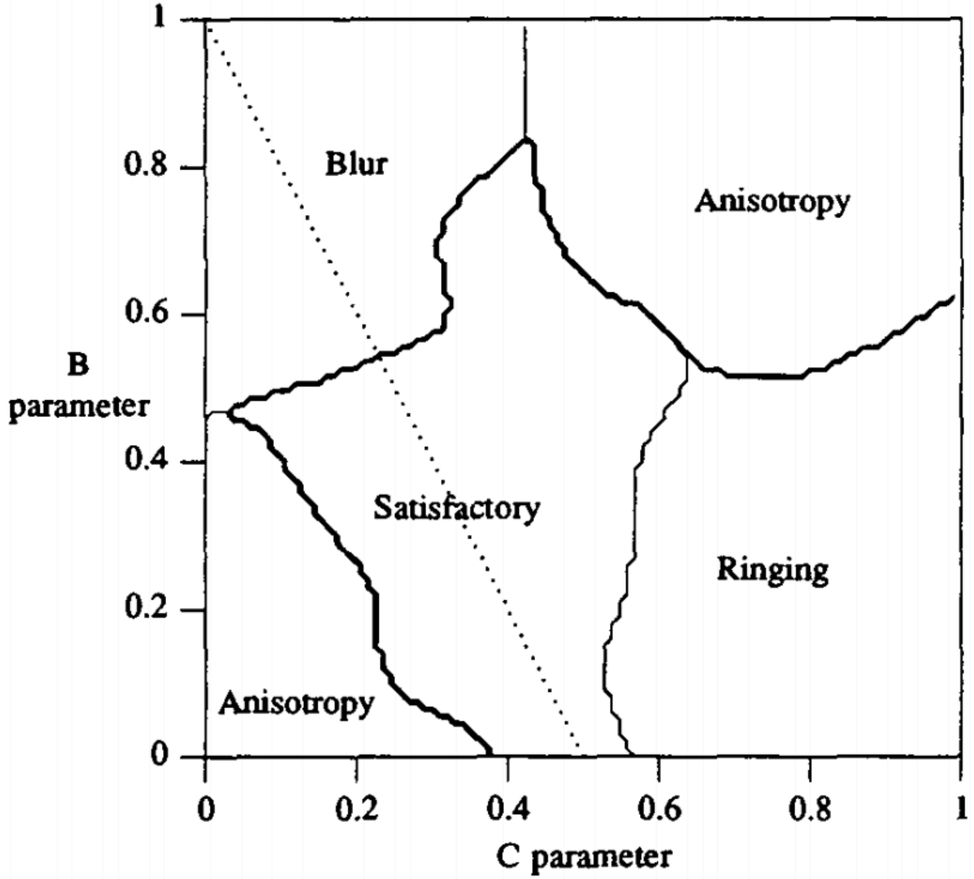


Figure 13. Regions of Dominant Subjective Behavior

Basic implementation

- **For each output pixel x, y**
  - Compute (fractional) inverse-transformed pixel coordinates $f^{-1}(x,y)$
    - For each input pixel x', y' in neighborhood of $f^{-1}(x,y)$
      - Compute kernel at $f^{-1}(x,y)$-(x', y')
      - Accumulate kernel * I(x', y')

- **Note: in general you can't pre-compute the kernel because the fractional values can be different for each pixel**
  - Although there are tricks

Kernel scaling

- Always the same for upsampling
- Proportional to scale factor for downsampling

In practice

- **separable (scale in x, then in y)**
  - Intermediate image of size (scale*width ; height) (anisotropically scaled)
- **basic implementation:**
  recompute weights for each pixel each neighbor
- **smarter: separable, precompute weights by row/column, take advantage of fractional or integer factors**

Bilinear



Bicubic (Photoshop)



Better reconstruction

- **Consider more than 4 pixels:**
  - bicubic, Lanczos, etc.
- **Try to sharpen edges**
- **Use training database of low-res/high-res pairs**
  - http://people.csail.mit.edu/billf/superres/index.html
- Deep learning

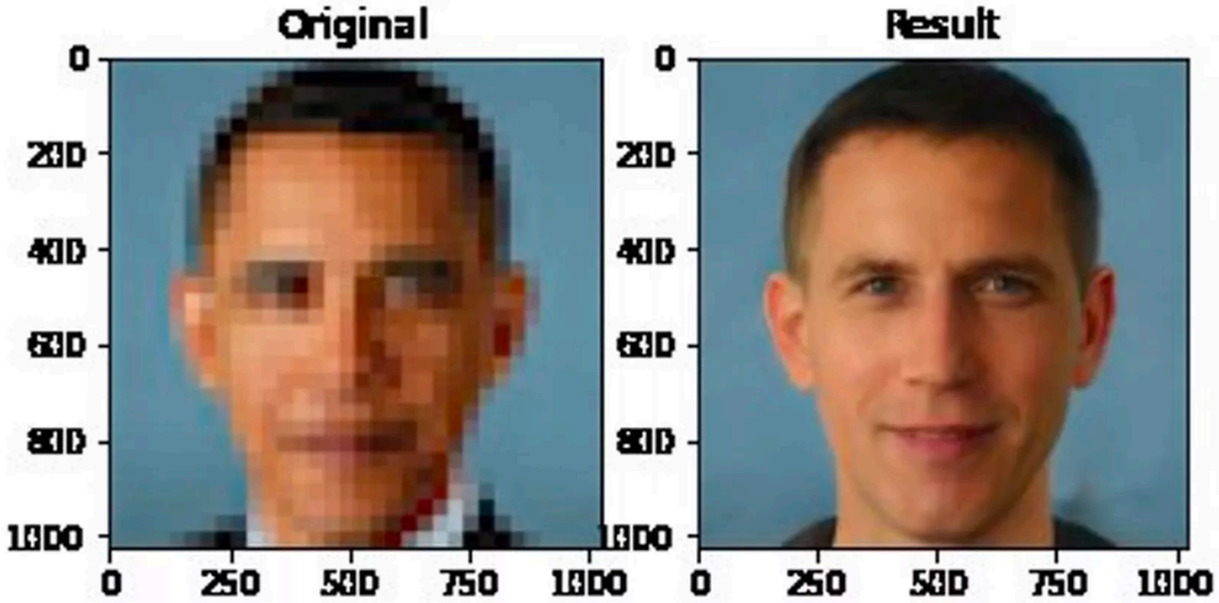
Bill Freeman's super-resolution





Better reconstruction

- Careful of fairness and bias when using ML
  - https://www.theverge.com/21298762/face-depixelizer-ai-machine-learning-tool-pulse-stylegan-obama-bias
  - https://thegradient.pub/pulse-lessons/

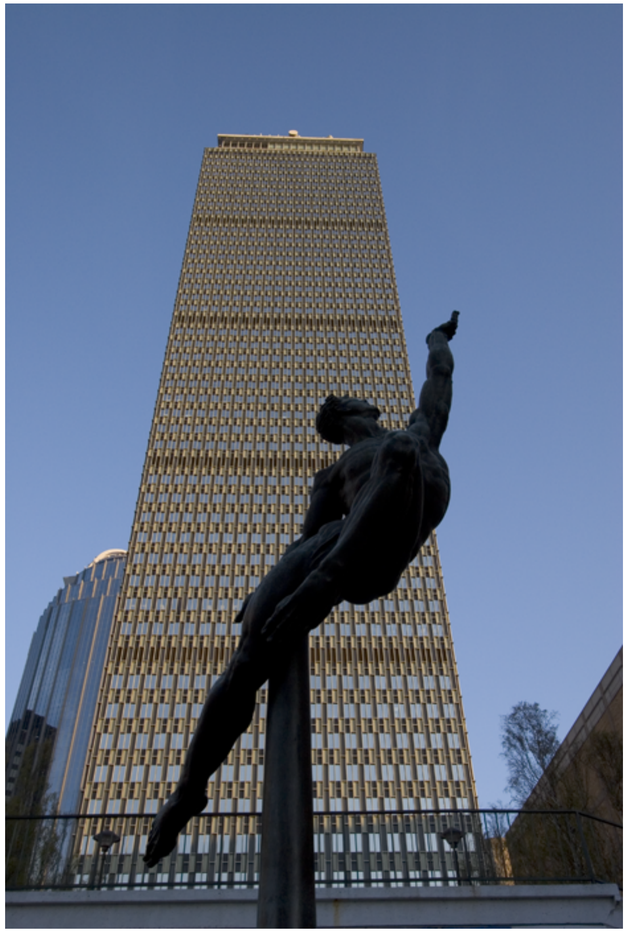


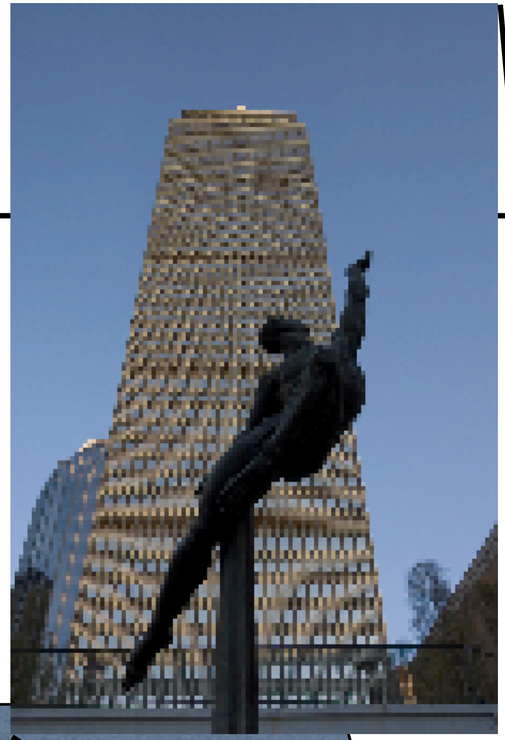
Questions?

Downsampling

Downsampling

- Using bilinear
- Yikes! Herringbone patterns



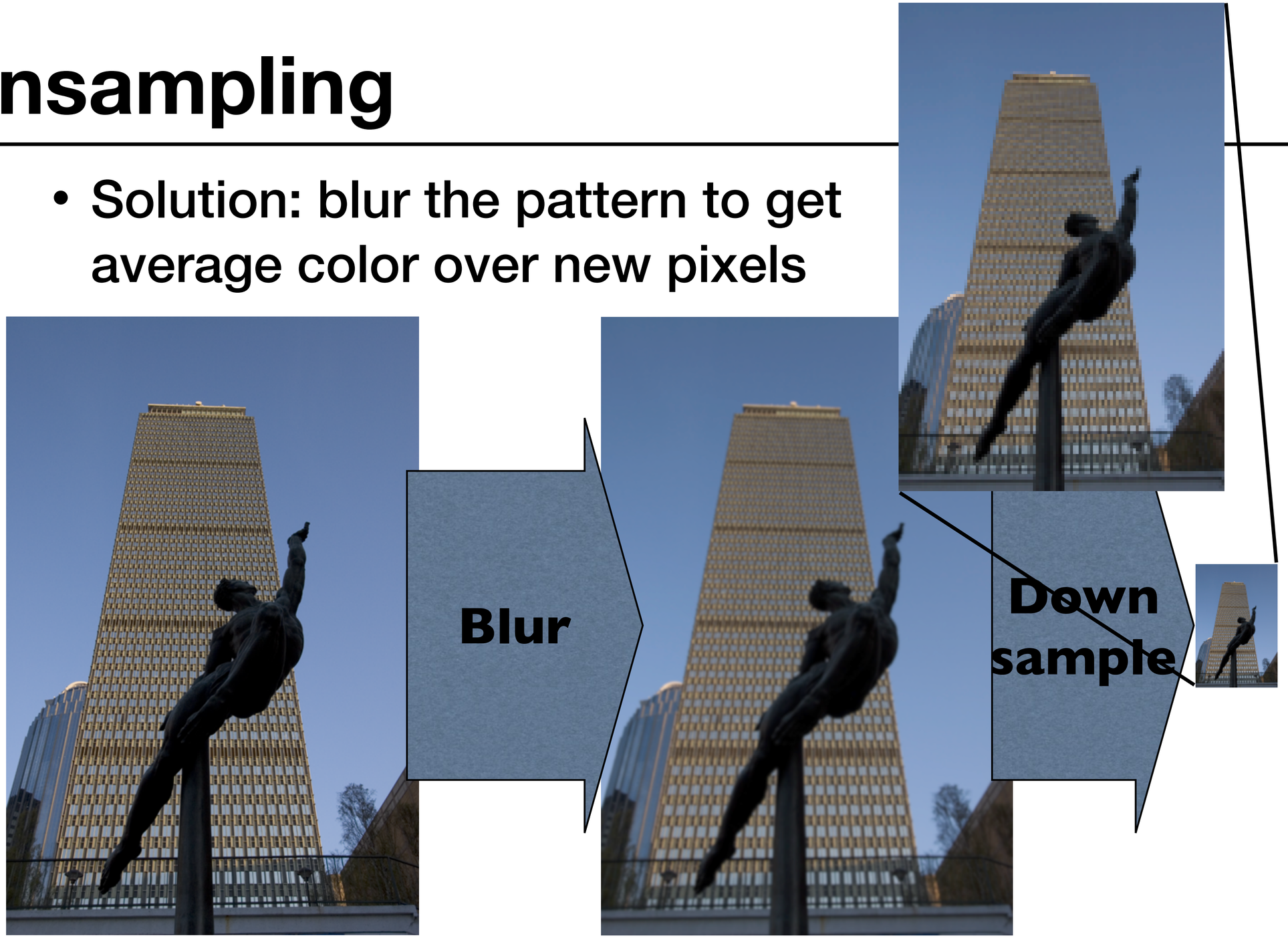


Downsample by
a scale of 0.2



Downsampling

- Solution: blur the pattern to get average color over new pixels



In practice

- Very similar to upsampling, usually same code
- Just scale kernel up by scaling factor
  - this means we average over more neighbors
- Think how far you need to gather
- For integer scale factor, can be precomputed

scipy is doing it wrong

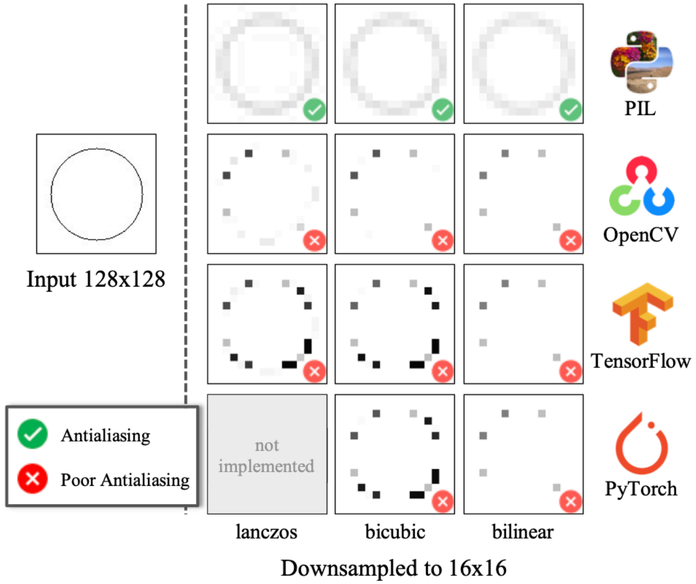
- https://arxiv.org/pdf/2104.11222.pdf
-

On Buggy Resizing Libraries and Surprising Subtleties in FID Calculation

Gaurav Parmar[1] Richard Zhang[2] Jun-Yan Zhu[1]
[1]Carnegie Mellon University [2]Adobe Research

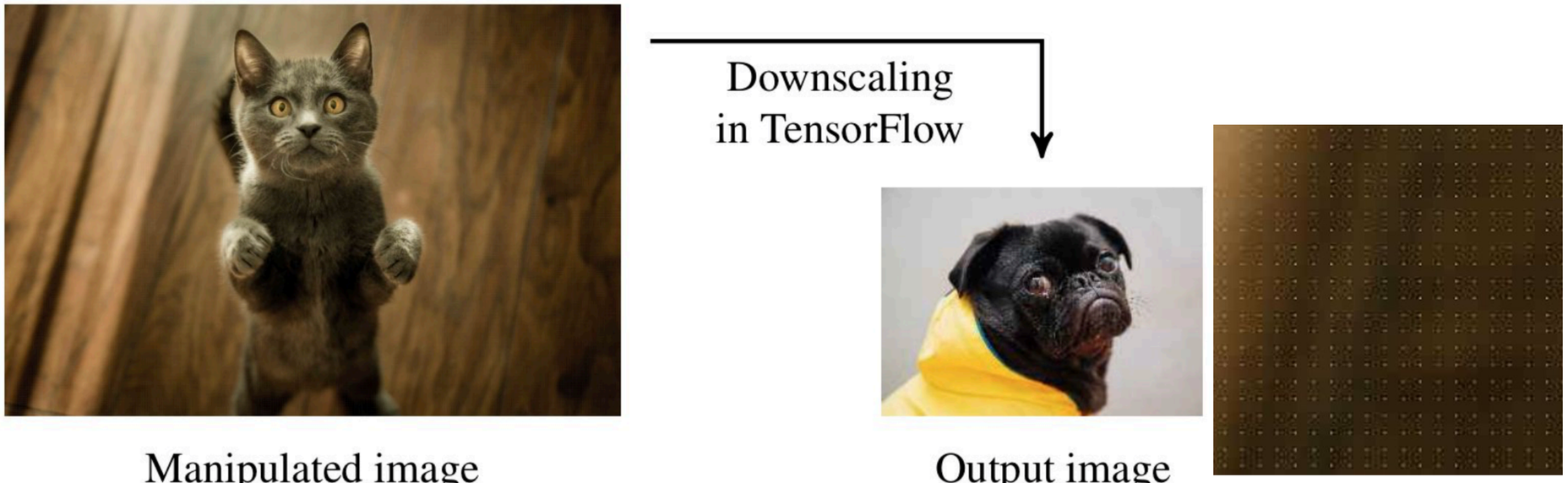
Abstract

We investigate the sensitivity of the Fréchet Inception Distance (FID) score to inconsistent and often incorrect implementations across different image processing libraries. FID score is widely used to evaluate generative models, but each FID implementation uses a different low-level image processing process. Image resizing functions in commonly-used deep learning libraries often introduce aliasing artifacts. We observe that numerous subtle choices need to be made for FID calculation and a lack of consistencies in these choices can lead to vastly different FID scores. In particular, we show that the following choices are significant: (1) selecting what image resizing library to use, (2) choosing what interpolation kernel to use, (3) what encoding to use



- **https://twitter.com/sigfpe/status/1457157479071199238**

- **https://twitter.com/AlexTamkin/status/1456149826337263621?fbclid=IwAR3_IH-3KwU3mf3eWfajZ0HiI-gIOAtCGkOxb5JkvzFUwfMCqqrg_PVdz80**



Manipulated image

Output image

Figure 1: Example of an image-scaling attack. Left: a manipulated image showing a cat. The scaling operation produces the right image with a dog.

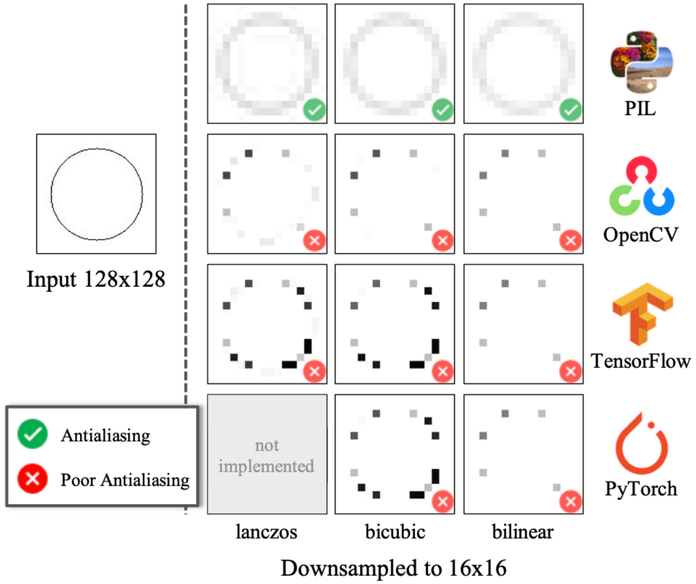
- https://arxiv.org/pdf/2104.11222.pdf
-

On Buggy Resizing Libraries and Surprising Subtleties in FID Calculation

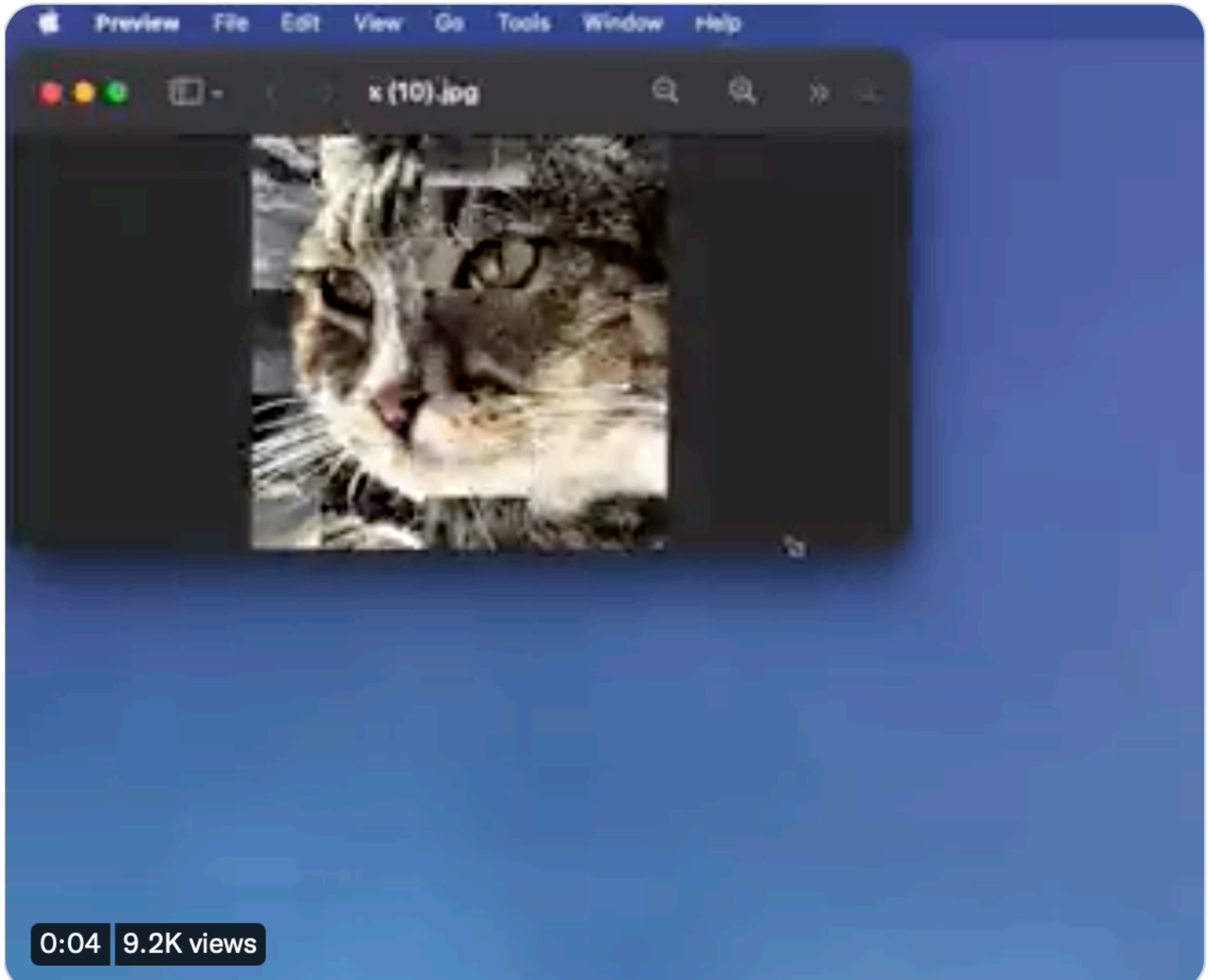
Gaurav Parmar[1] Richard Zhang[2] Jun-Yan Zhu[1]
[1]Carnegie Mellon University [2]Adobe Research

Abstract

We investigate the sensitivity of the Fréchet Inception Distance (FID) score to inconsistent and often incorrect implementations across different image processing libraries. FID score is widely used to evaluate generative models, but each FID implementation uses a different low-level image processing process. Image resizing functions in commonly-used deep learning libraries often introduce aliasing artifacts. We observe that numerous subtle choices need to be made for FID calculation and a lack of consistencies in these choices can lead to vastly different FID scores. In particular, we show that the following choices are significant: (1) selecting what image resizing library to use, (2) choosing what interpolation kernel to use, (3) what encoding to use



- https://twitter.com/sigfpe/status/1457157479071199238



More info

- http://web.stanford.edu/class/cs448f/lectures/1.2/Sampling.pdf
- https://en.wikipedia.org/wiki/Bicubic_interpolation
- http://entropymine.com/imageworsener/bicubic/
- http://entropymine.com/imageworsener/resample/
- separable!
- upsampling is same, just clamp kernel scale factor to 1.0
- downsampling not super easy to debug

Advice by Andrew Adams

I don't know of a good general reference, but I definitely have opinions on the right way to implement downsampling.
https://github.com/halide/Halide/blob/master/apps/resize/resize_generator.cpp

Downsampling is linear and separable, so the general idea is to explicitly compute the sparse matrix that downsamples a single row/col, then apply it to each row/col. The size of the sparse matrix is tiny relative to the image, so computing the weights and handling boundary conditions becomes approximately free.

In more detail to downsample in x:

1) For each x coordinate in the output, translate and scale the continuous downsampling kernel appropriately to have the right footprint on the input.

2) Sample that translated, scaled, continuous kernel at all integer locations on the x axis of the input space where it has non-zero values. You can compute closed-form expressions for the index of the first non-zero value and the maximum number of non-zero values, so it's easy to represent the sparse matrix as a simple max_kernel_taps x input_width table.

3) Renormalize each row of the matrix to sum to 1. I think you can avoid this if you're careful with the kernel, but this is also convenient way to handle boundary conditions if you want to just truncate the kernel at the image edges.

4) Optionally quantize the matrix and store it as integers that sum to a power of 2. Do this if your image is low precision and the machine you're on has high throughput for integer multiply-adds (e.g. a DSP). The example code doesn't do this. Forcing it to sum to a power of 2 is more complicated that just quantizing scaled values - you need error diffusion.

5) Apply this matrix to each row of the image. The bulk of the runtime is here.

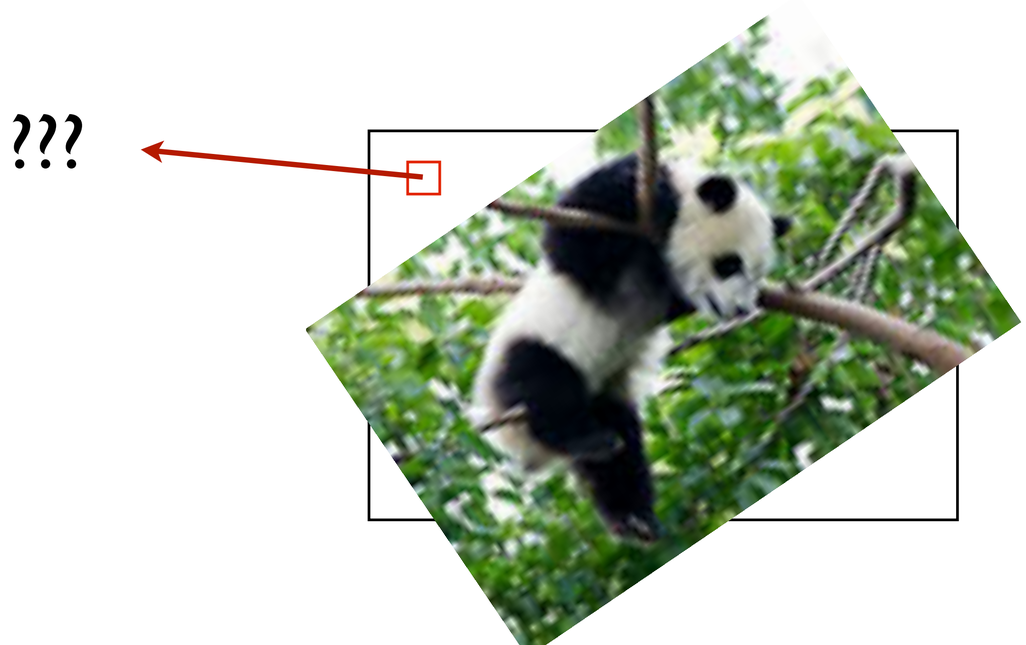
You definitely want to downsample in y first, because that vectorizes nicely. Then you have less data to deal with for the downsample in x. If the downsampling factor is very large, you can vectorize the downsample in x along the reduction over the kernel, but if the downsample is large then the x pass takes nearly no time relative to the y pass, so it's not that important. You can also downsample in y, transpose, downsample in y, transpose, but I suspect it would be slower.

For rational downsampling factors the sparse matrix becomes periodic, but it's not really worth taking advantage of this - the matrix is likely to fit in cache so you're not going to be bandwidth-limited reading values from it.

Padding

Padding problems

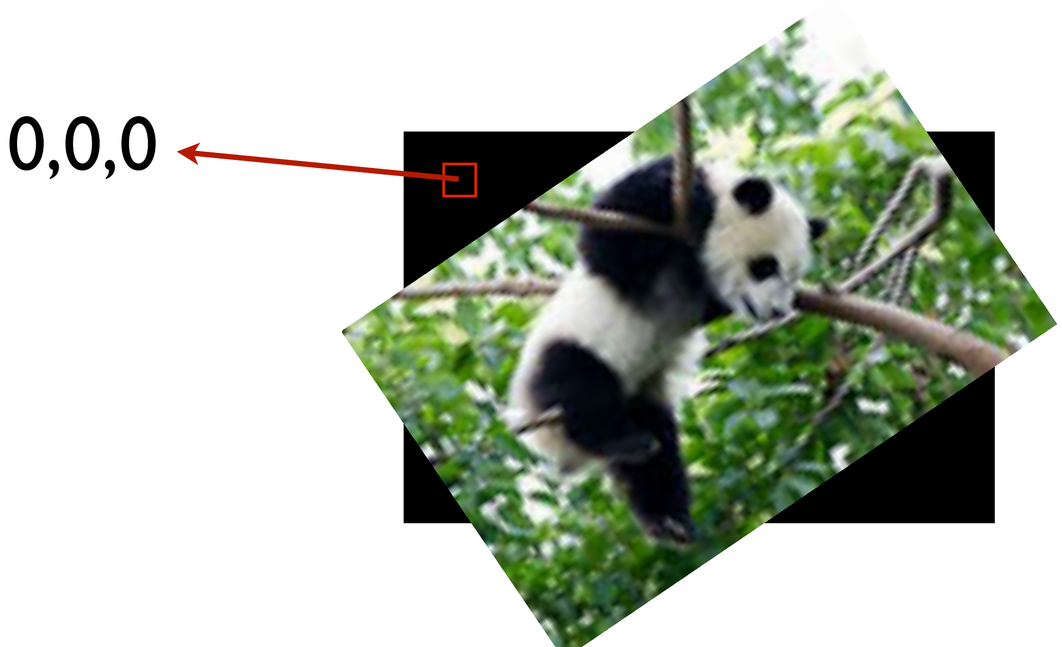
- **Sometimes, we try to read outside the image**
  - e.g. y, x are negative
  - For example, we try to rotate an image



Black Padding

```
def getBlackPadded(im, y, x):
    if (x<0) or (x>=im.shape[1]) or (y<0)
        or (y>= im.shape[0]):
        return numpy.array([0, 0, 0])
    else: return im[y, x]
```

•



Edge Padding

```
def clipX(im, x): return min(width(im)-1, max(x, 0))
def clipY(im, y): return min(height(im)-1, max(y, 0)
def getSafePix(im, y, x):
    return im[clipY(im, y), clipX(im, x)]
```



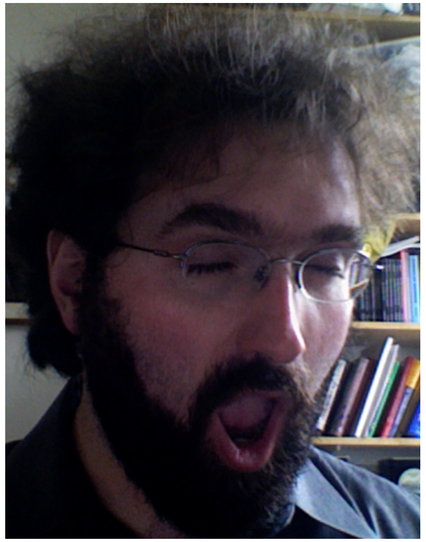
Questions?

Warping & Morphing

Important scientific question

- **How to turn Dr. Jekyll into Mr. Hyde?**
- **How to turn a man into a werewolf?**

- **Powerpoint cross-fading?**



Important scientific question

- **How to turn Dr. Jekyll into Mr. Hyde?**
- **How to turn a man into a werewolf?**

- **Powerpoint cross-fading?**



From An American Werewolf in London

- **or**
- **Image Warping and Morphing?**

Digression: old metamorphoses

- http://en.wikipedia.org/wiki/The_Strange_Case_of_Dr._Jekyll_and_Mr._Hyde
- http://www.eatmybrains.com/showtopten.php?id=15
- http://www.horror-wood.com/next_gen_jekyll.htm
- **Unless I'm mistaken, both employ the trick of making already-applied makeup turn visible via changes in the color of the lighting, something that works only in black-and-white cinematography. It's an interesting alternative to the more familiar Wolf Man time-lapse dissolves. This technique was used to great effect on Fredric March in Rouben Mamoulian's 1932 film of *Dr. Jekyll and Mr. Hyde*, although Spencer Tracy eschewed extreme makeup for his 1941 portrayal.**



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1941

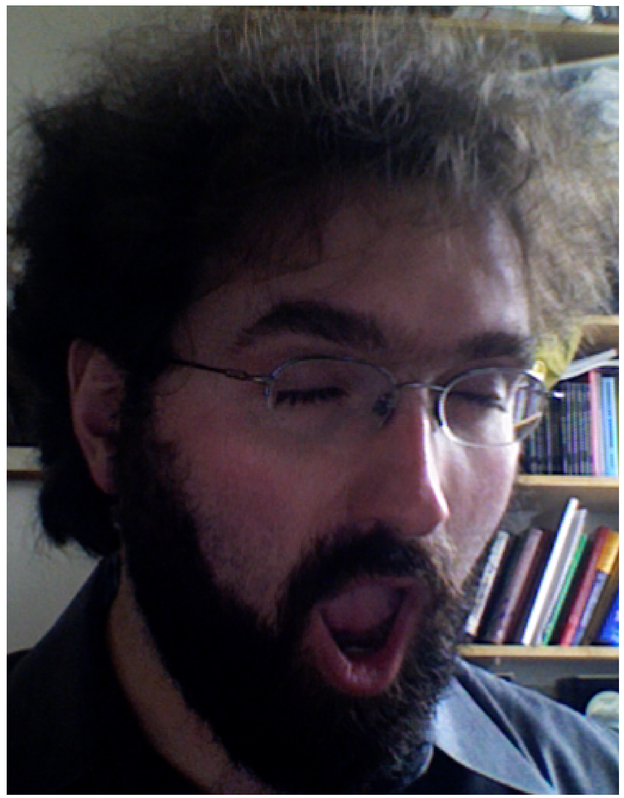


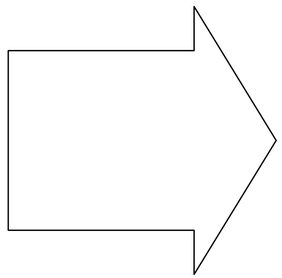
- https://filmschoolrejects.com/sh-the-octopus-transformation/
- https://x.com/rainmaker1973/status/1983064162583368154?s=46&t=aUZMRTg07VYQGqu7K-19Lw
- https://filmschoolrejects.com/sh-the-octopus-transformation/
-



Challenge

- **“Smoothly” transform a face into another**
- **Related: slow motion interpolation interpolate between key frames**



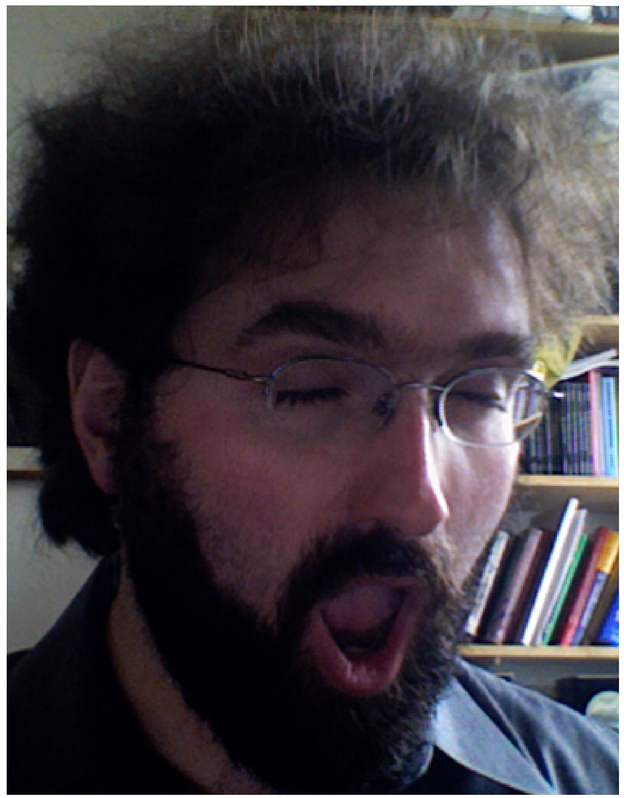
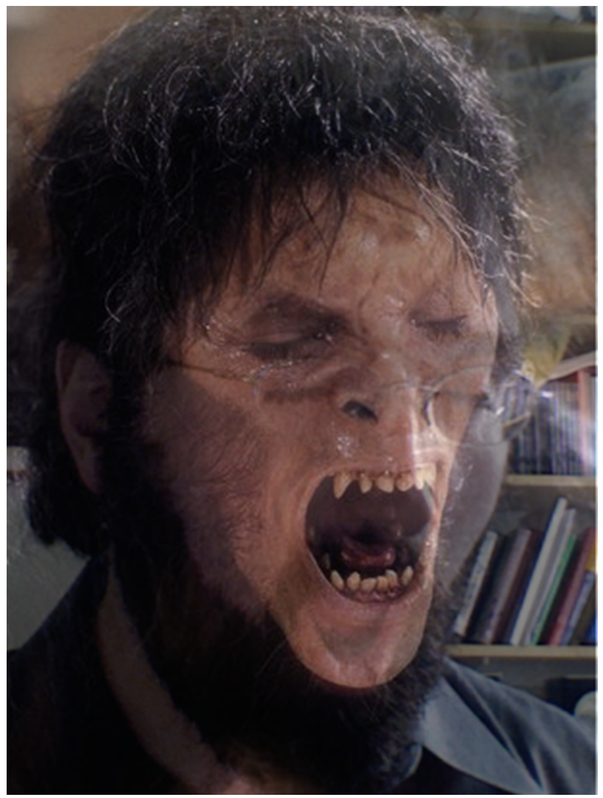




Averaging images

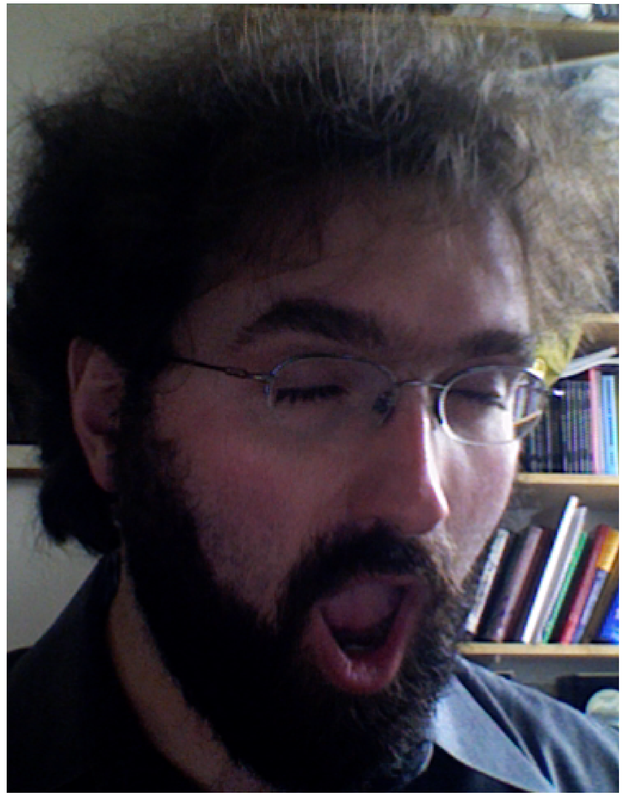
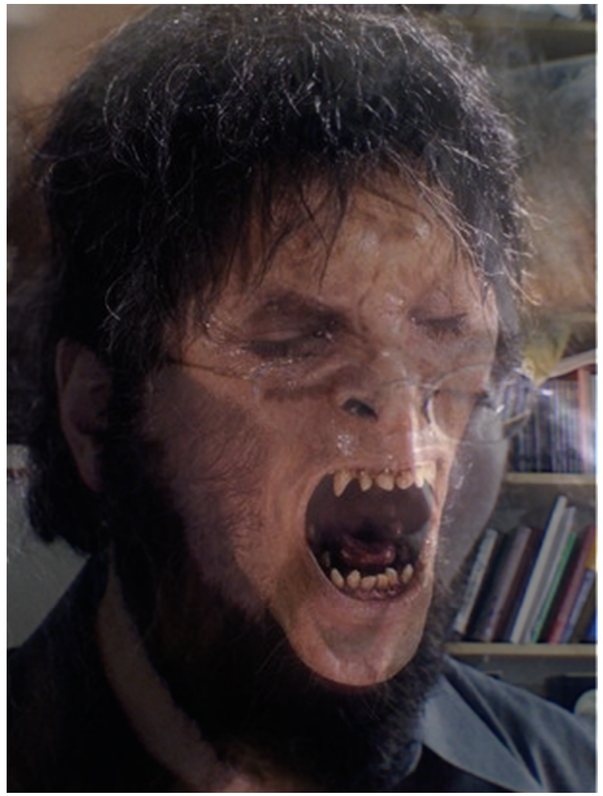
- **Cross-fading**

  output[y, x]=t * im1[y, x] +(1-t) * im2[y, x]

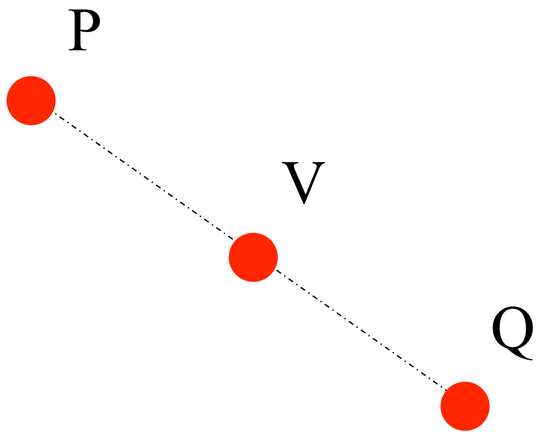

Problem with cross fading

- **Features (eyes, mouth, etc) are not aligned**
- **It is probably not possible to get a global alignment**
- **We need to interpolate the LOCATION of features**
  – domain transform!

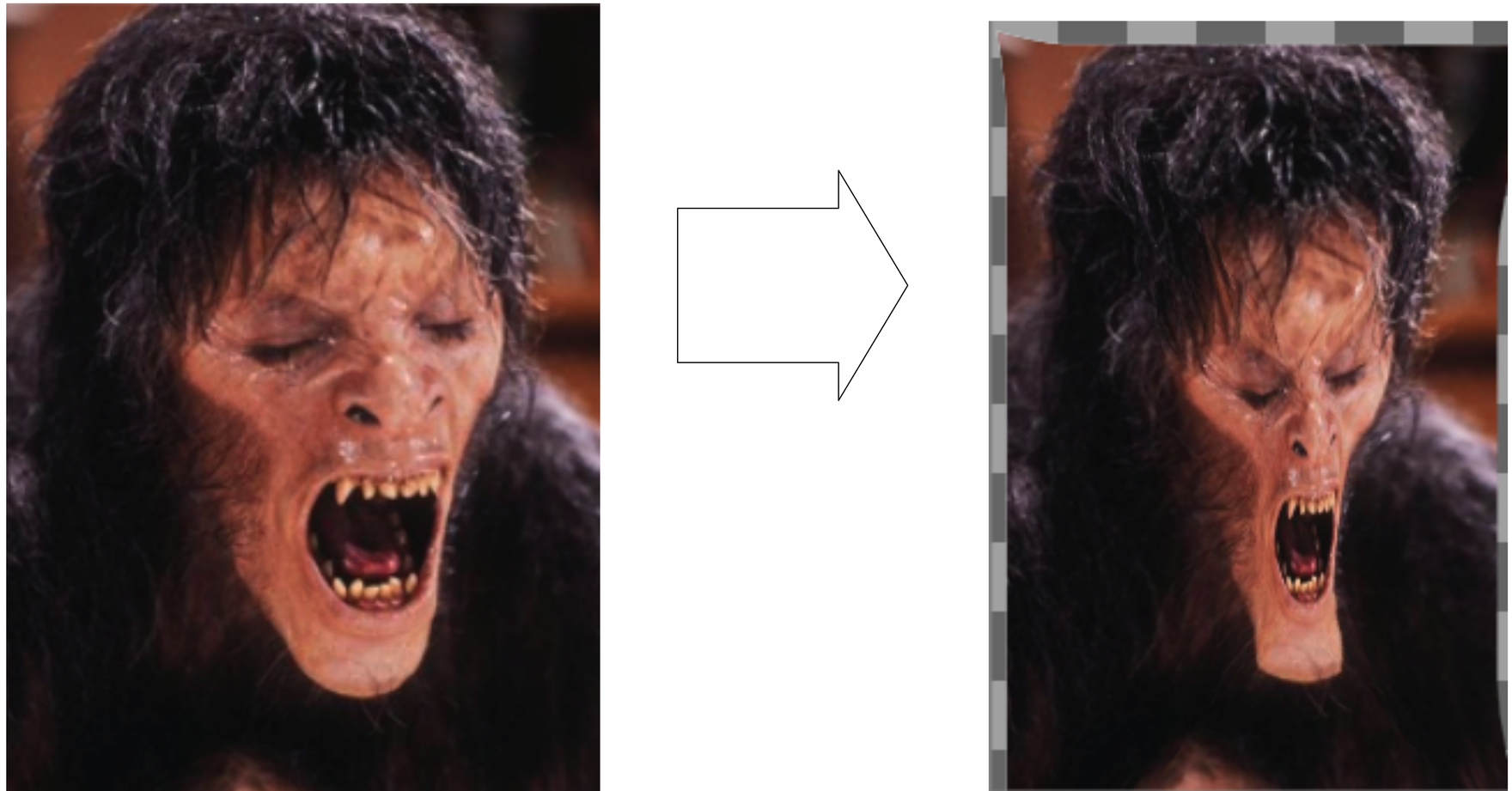

Averaging points (location)

- **P & Q are two 2D points (in the “domain”)**
- **V= t P + (1-t) Q**



Warping

- **Move pixel spatially: C’(x,y)=C(f(x,y))**
- **Leave colors unchanged**



Warping

- **Deform the domain of images (not range)**
- **Central to morphing**
- **Also useful for**
  - Optical aberration correction
  - Video stabilization
  - Slimming people down
  - ...



original image



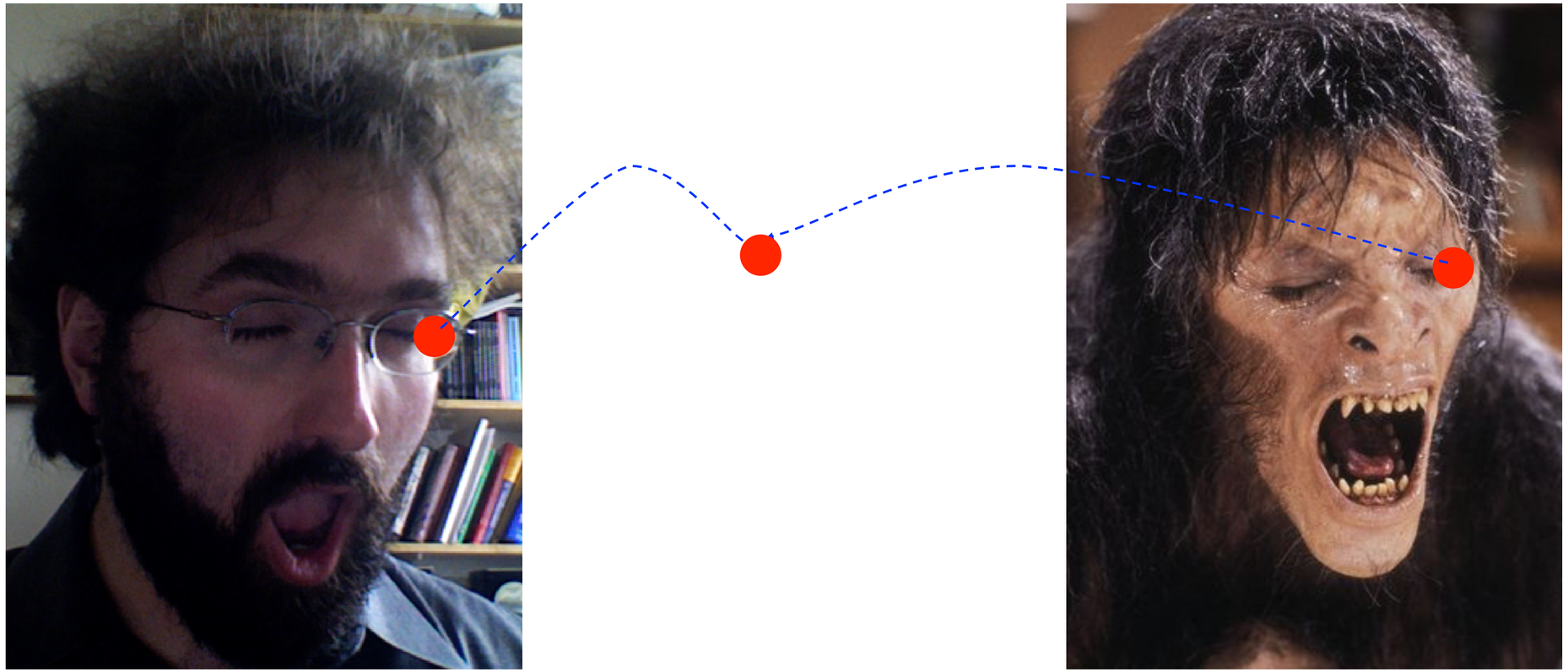
DxO Optics Pro correction

Recap & questions

- **Color (range) interpolation:**
  output[y, x]=t * im1[y, x] +(1-t) * im2[y, x]
- **Location (domain) interpolation: V= t P + (1-t) Q**
- **Warping: domain transform out(x,y)=im(f^{-1}(x,y))**

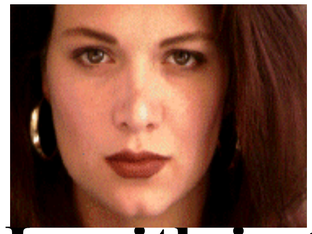
Morphing: combine both

- **For each pixel**
  - Transform its location like a vector (domain)
  - Then linearly interpolate colors (range)



Morphing

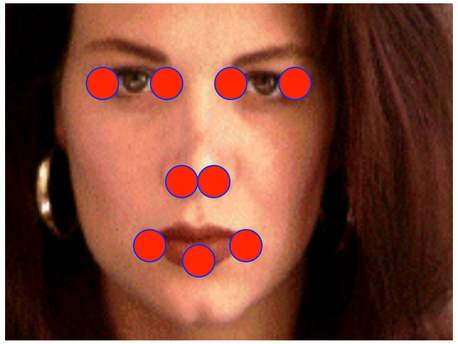
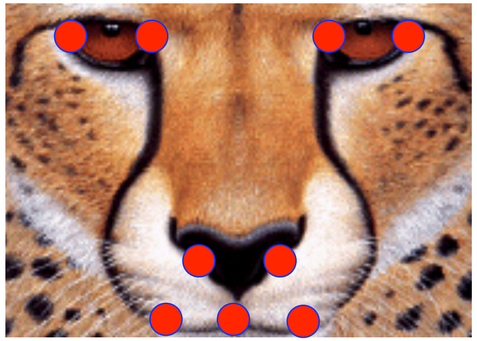
- **Input: two images I_0 and I_{N+1}**




- **Expected output: image sequence I_i, with i∈ 1..N**

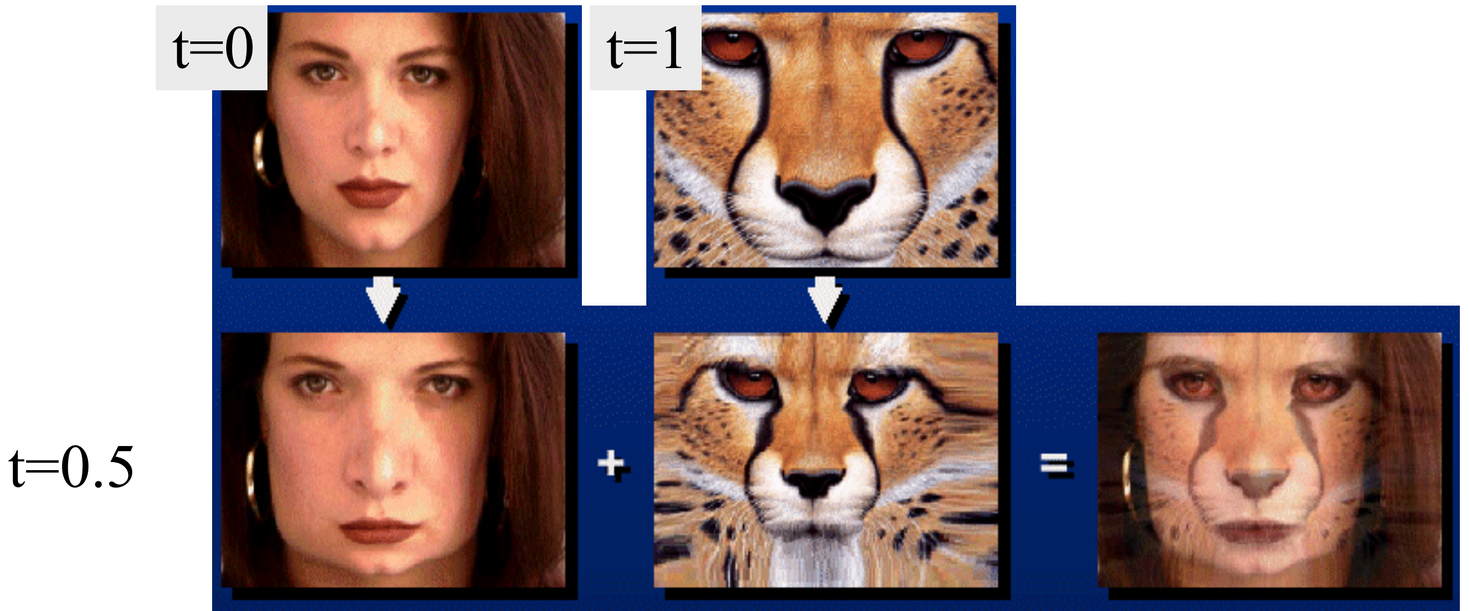


- **User specifies sparse correspondences on the images**

Morphing

- **For each intermediate frame I_t**
  - Interpolate feature locations $P^t_i = (1- t)\, P^0_i + t\, P^1_i$
  - Perform **two** warps: one for I_0, one for I_1
    - Deduce a dense warp field from the pairs of features
    - Warp the pixels
  - Linearly interpolate the two warped images



Warping

How do we specify the warp?

- **Before, we saw simple transformations**
  – linear, affine, perspective



translatio
n



rotation



aspect



affine



perspective

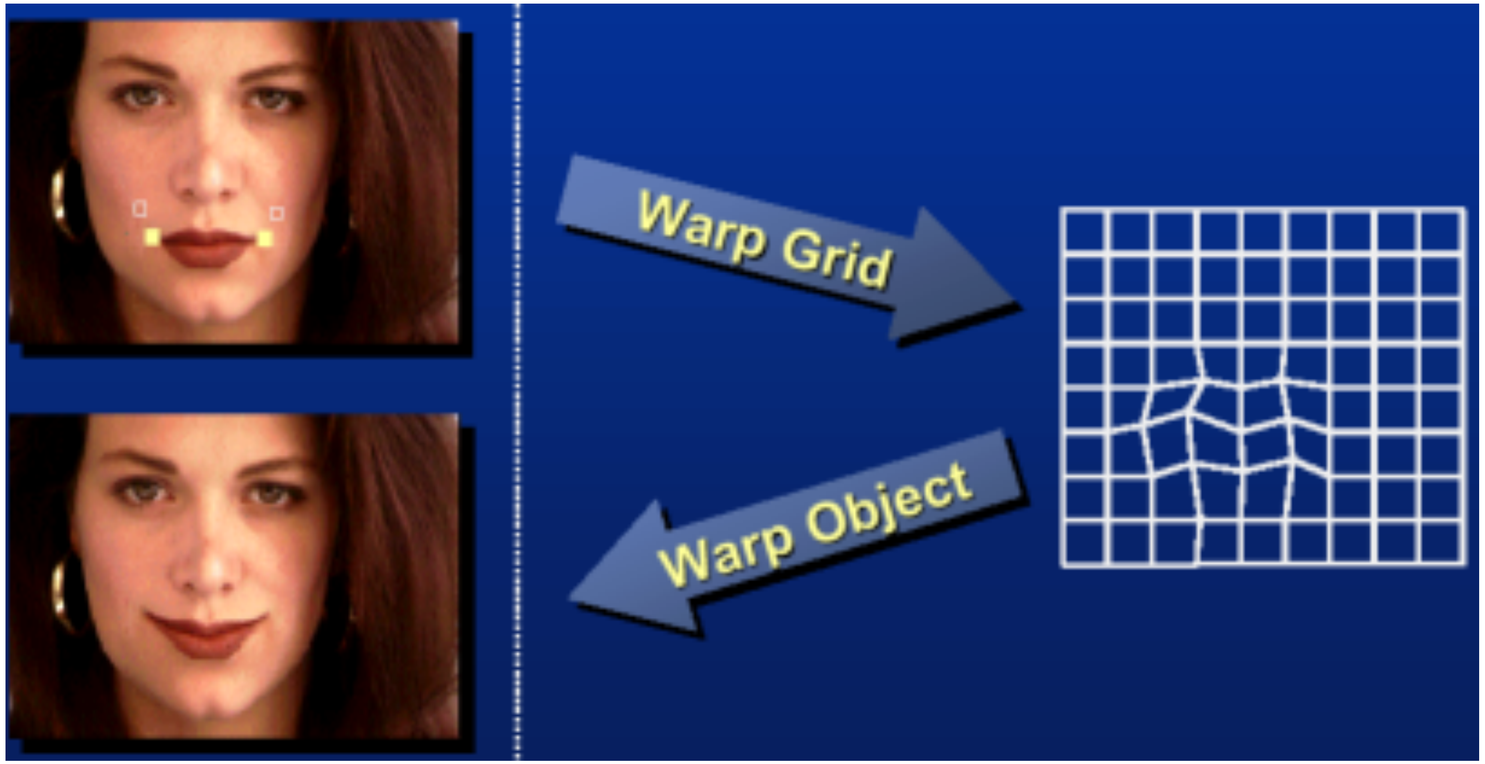


cylindrical

- **But we want more flexibility**

Image Warping – parametric

- **Move control points to specify a spline warp**
- **Spline produces a smooth vector field**



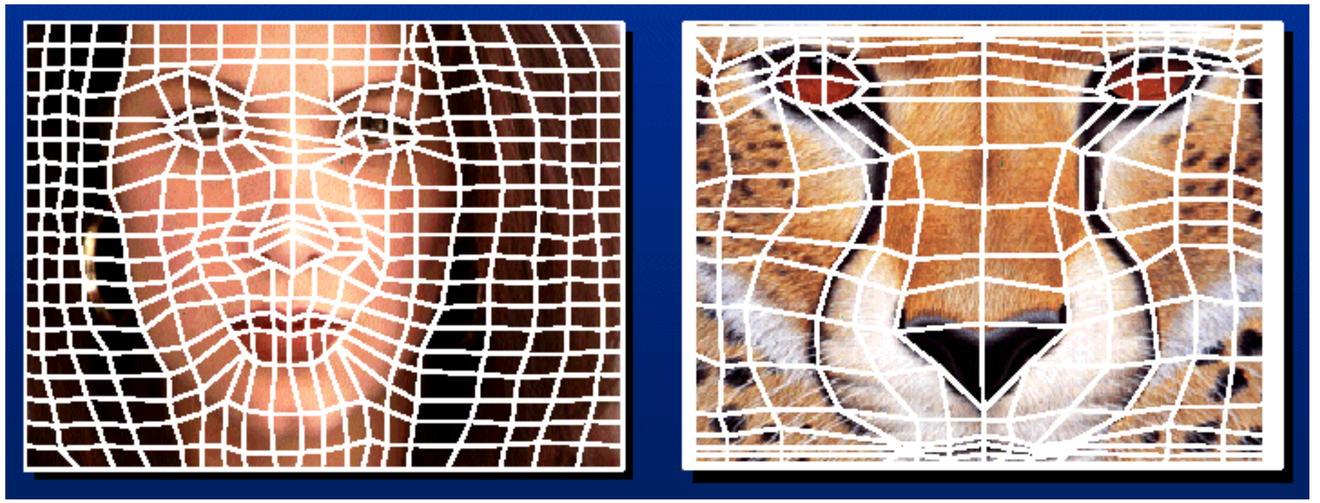
Slide Alyosha Efros

Warp specification - dense

- **How can we specify the warp?**

  Specify corresponding *spline control points*

  - *interpolate* to a complete warping function



But we want to specify only a few points, not a grid

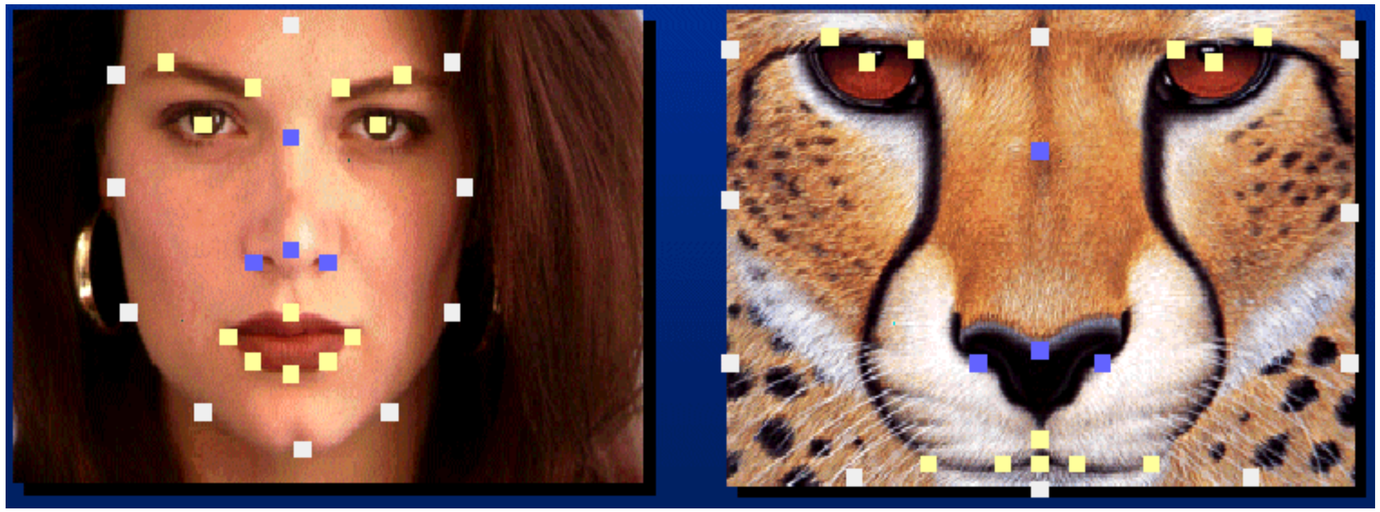
Slide Alyosha Efros

Warp specification - sparse

- **How can we specify the warp?**

  Specify corresponding *points*

  - *interpolate* to a complete warping function
  - How do we do it?

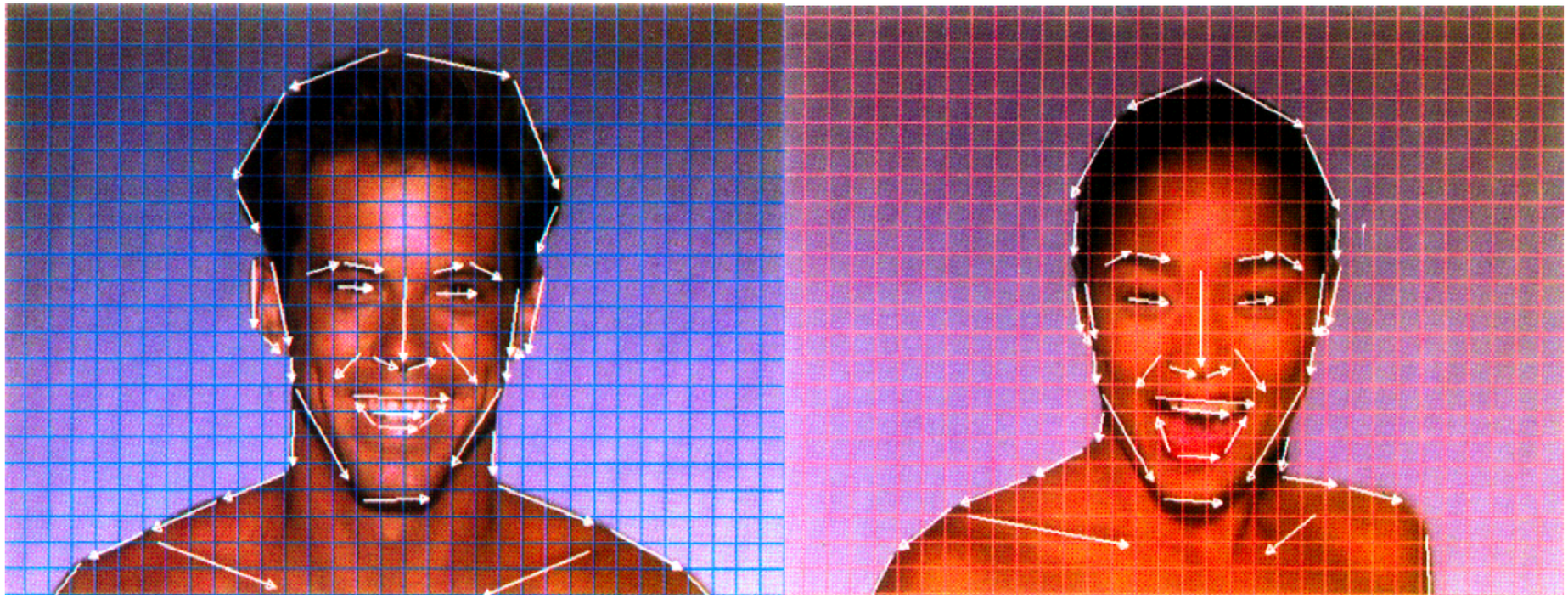


How do we go from feature points to pixels?

Slide Alyosha Efros

Beier and Neely

- **Specify warp based on pairs of segments**
  - http://dl.acm.org/citation.cfm?id=134003
  - Feature-Based Metamorphosis, SIGGRAPH 1992
  - Used in Michael Jackson' Black and White Music Video
  - Pset 2!!

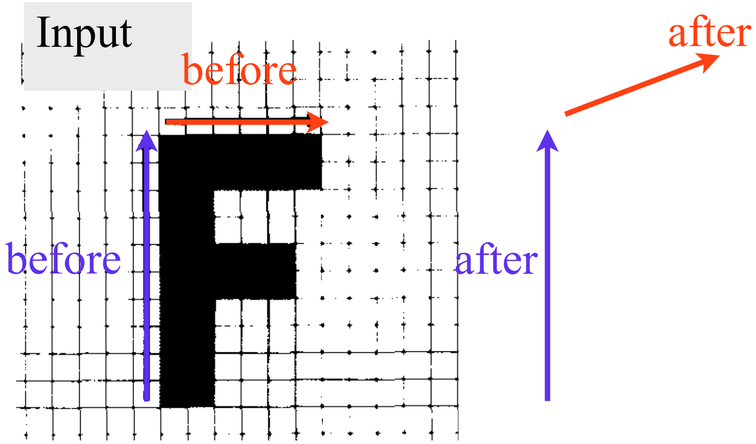


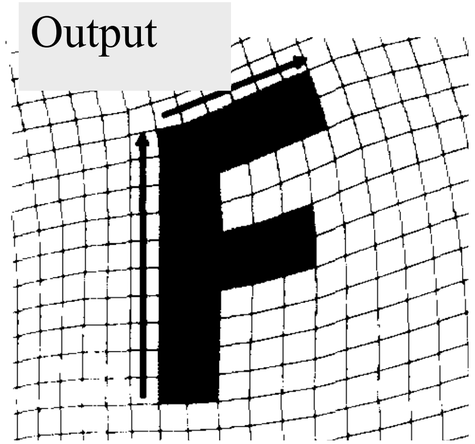
Questions?

Segment-based warping

Problem statement

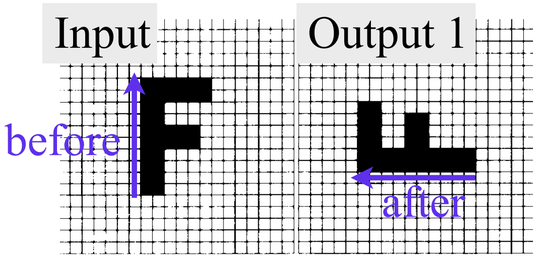
- **Inputs: One image, two lists of segments before and after, in the image domain**
- **Goal: warp the image "following" the displacement of the segments**





Idea

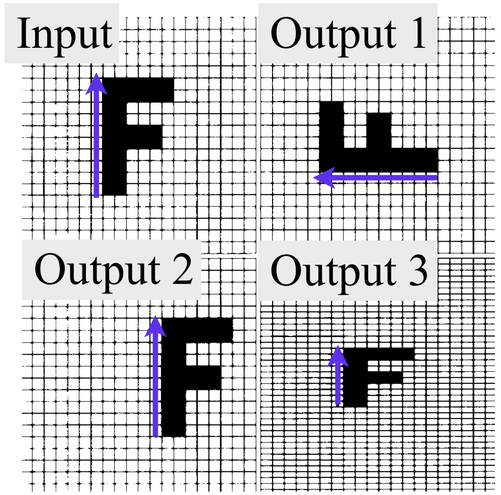
- **Each before/after pair of segment implies a planar transformation**
  - simple and affine



Single line transforms

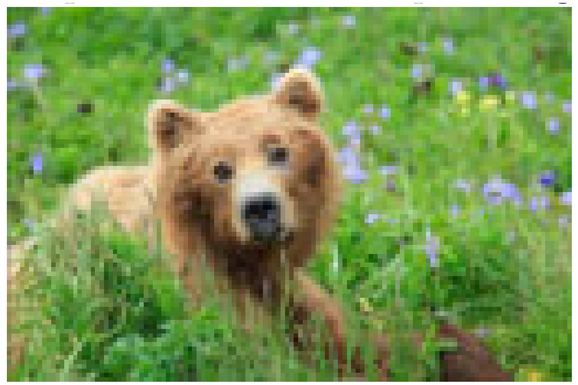
Idea

- **Each before/after pair of segment implies a planar transformation**
  - simple and affine

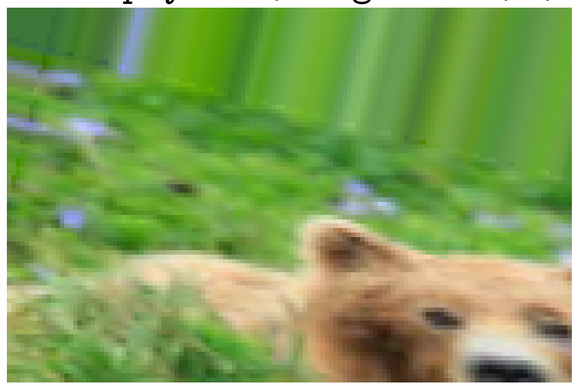


Single line transforms

Test

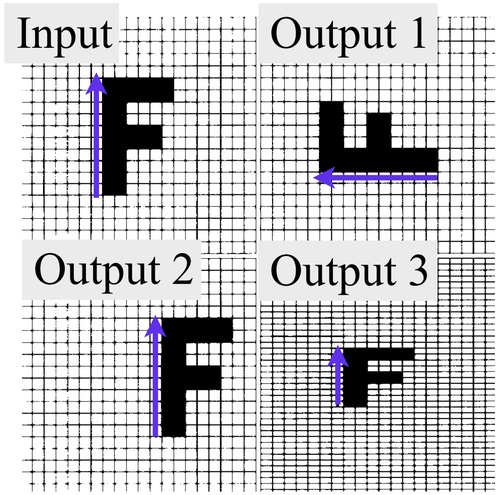


→ `warpBy1(im, segment(0,0, 10,0), segment(10, 10, 30, 15))` →

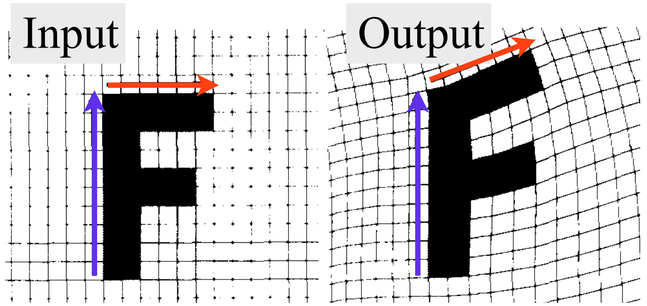


Idea

- **Each before/after pair of segment implies a planar transformation**
- **Then take weighted average of transformations**



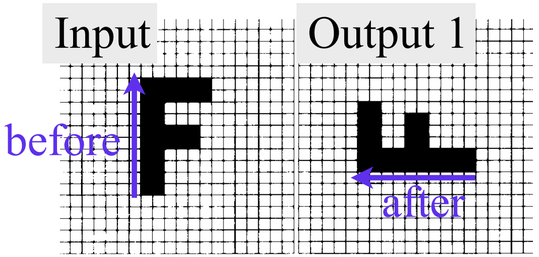
Single line transforms



Transform wrt 2 lines

Back to single segment

- **Each before/after pair of segment implies a planar transformation**
  - simple and affine



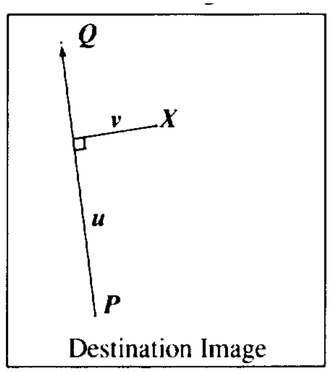
Single line transforms

Linear algebra refresher



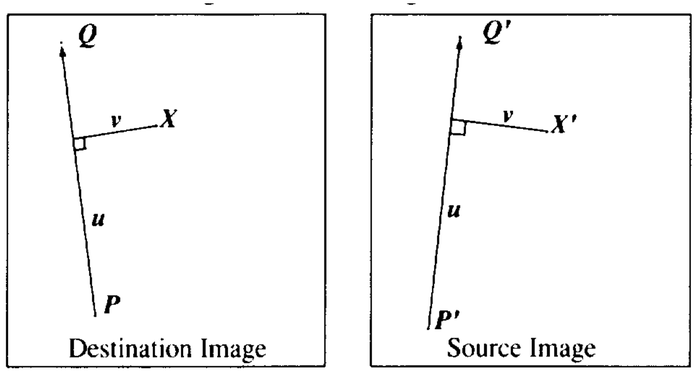
Transform wrt 1 segment

- **Define a coordinate system with respect to segment**
  - 1 dimension, *u*, along segment
  - 1 dimension, *v*, orthogonal to segment



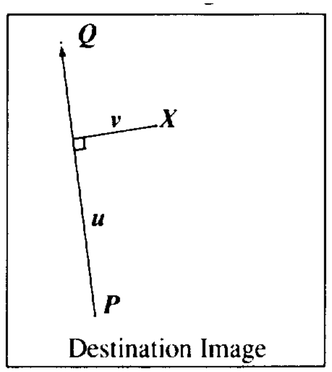
Transform wrt 1 segment

- **Define a coordinate system with respect to segment**
  - 1 dimension, *u*, along segment
  - 1 dimension, *v*, orthogonal to segment
- **Compute u, v in one image**
  - The after one, because we use the inverse transform
- **Compute point corresponding to u, v in second image**



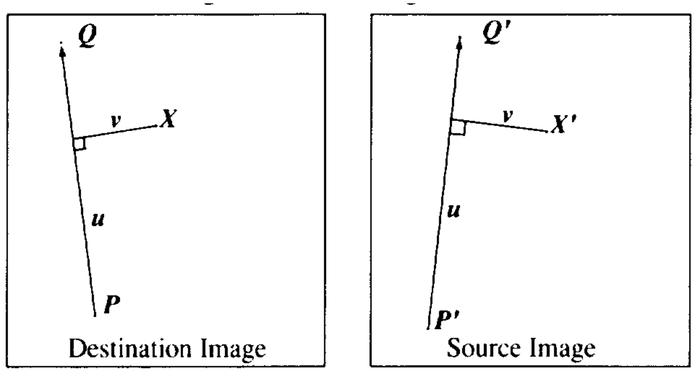
Computing u, v

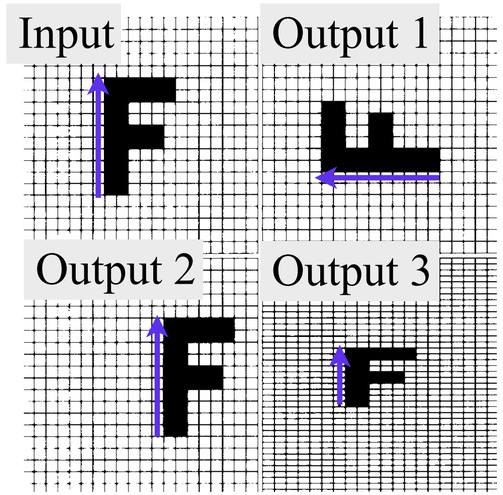
- **u = PX.PQ/||PQ||²**
  - this way u is 0 at P and 1 at Q
- **v=PX.perpendicular(PQ)/||PQ||**
  - where perpendicular(PQ) is PQ rotated by 90 degrees, and has length ||PQ||
  - unlike u which is normalized, v is in distance units



Transforming a point given u, v

- **X'=P'+uP'Q+v perpendicular(P'Q')/||P'Q'||**
- **The u component is scaled according to segment scaling**
- **But v is absolute (see output3)**
  - They say they tried to scale v as well but it didn't work as well





Single line transforms

Questions?

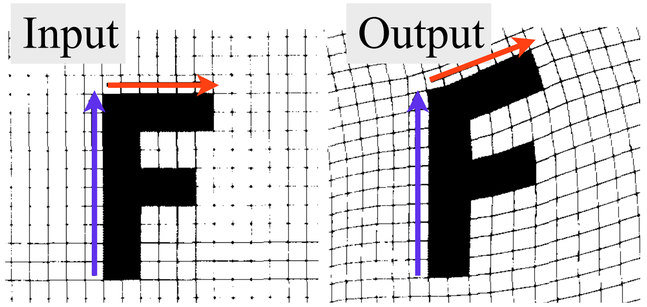


Multiple segments

- **For each point X**
  - For each segment pair s_before[i], s_after[i]
    - Transform X into X'i according to this pair
  - Compute weighted average of all transformed X'i
    - weight according to distance to segments

$$weight = \left(\frac{length^p}{a + dist} \right)^b$$

where a, b, p
control the influence



Transform wrt 2 lines

Distance to a segment

- **Multiple cases...**

  – dot product, test >0, <1

Debugging: example

- **Debugging my distance function**



Questions?

- **More debugging**

