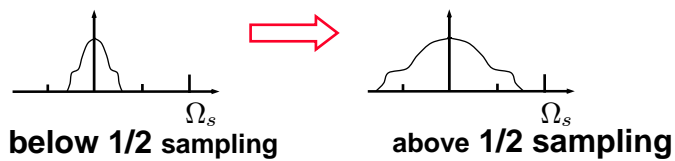

Image processing II

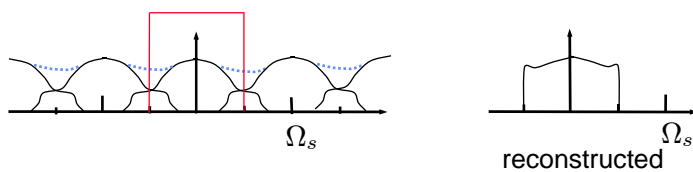
© 2001, Denis Zorin

Shrinking: problem

Shrinking by a factor a
in freq. domain becomes stretching by a



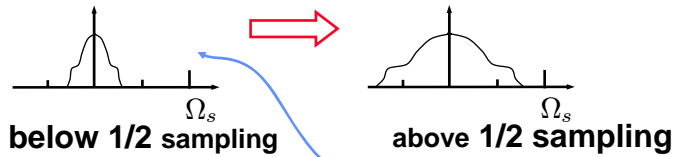
Won't be able to reconstruct correctly
= won't see the expected image



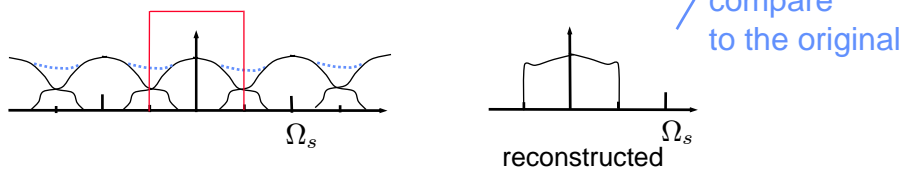
© 2001, Denis Zorin

Shrinking: problem

Shrinking by a factor $a < 1$
in freq. domain becomes stretching by $1/a$



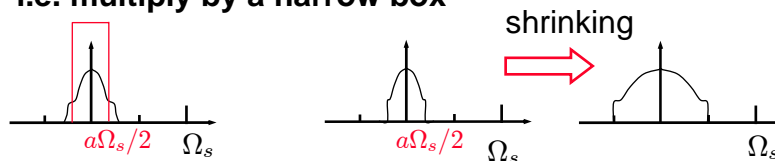
Won't be able to reconstruct correctly
= won't see the expected image



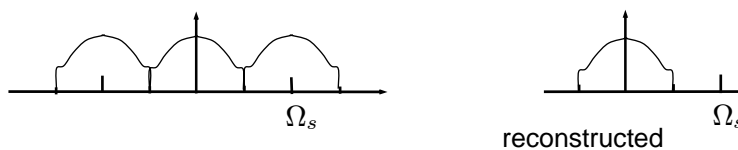
© 2001, Denis Zorin

Shrinking: solution

Theoretical solution:
BEFORE shrinking, remove high frequencies,
i.e. multiply by a narrow box



Now, there is no overlap, can reconstruct
(= see the right thing)



© 2001, Denis Zorin

Image shrinking

For simplicity, derive everything in 1D; shrinking of a two dimensional image is done in two steps: first in x direction, then in y direction.

Problem: after shrinking have too few pixels to represent all pixels of the original.

Solution: do local averaging; similar to the continuous case, but instead of integration do summation.

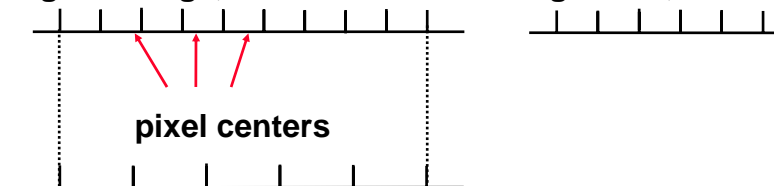
© 2001, Denis Zorin

Image shrinking

Shrinking by a factor $a < 1$:

original image, size w

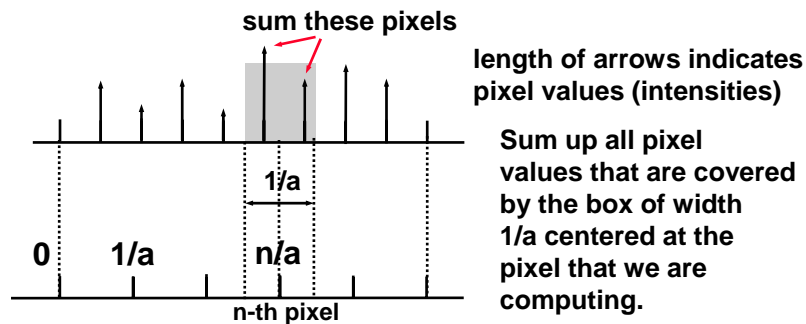
target size, aw



think about pixels of the target as “fat pixels”; the size of a fat pixel is $1/a$; the the size of a the (rescaled) target image is $aw(1/a) = w =$ size of the original (but the pixel size is different!)

© 2001, Denis Zorin

Image shrinking



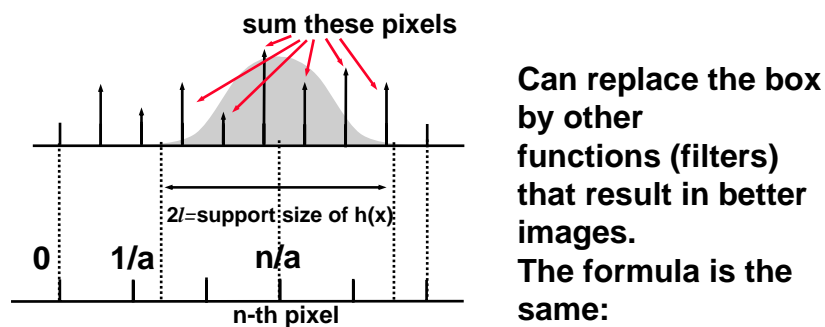
If box function of width 1 is $h(x)$, then:

- box function of width $1/a$ is $h(ax)$
- box function of width $1/a$ centered at n/a is $h(a(x-n/a)) = h(ax-n)$
- add a scale factor a , so that we do not get high intensities:

$$p^{\text{shrunked}}[n] = \sum_i h(ai - n)p[i]$$

© 2001, Denis Zorin

Image shrinking



$$p^{\text{shrunked}}[n] = \sum_i h(ai - n)p[i]$$

We have to sum only over pixels for which $h(ai - n)$ is not zero. If $h(x)$ is zero outside $[-l, l]$, The range for i is determined from

$$-l \leq ai - n \leq l$$

© 2001, Denis Zorin

Image shrinking

For simplicity, derive everything in 1D; shrinking of a two dimensional image is done in two steps: first in x direction, then in y direction.

Problem: after shrinking have too few pixels to represent all pixels of the original.

Solution: do local averaging; similar to the continuous case, but instead of integration do summation.

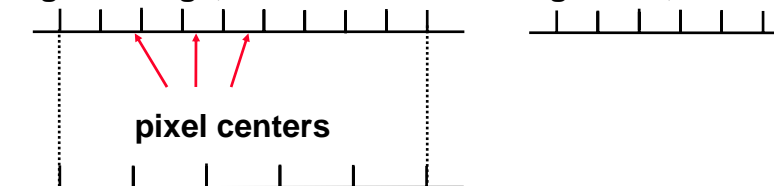
© 2001, Denis Zorin

Image shrinking

Shrinking by a factor $a < 1$:

original image, size w

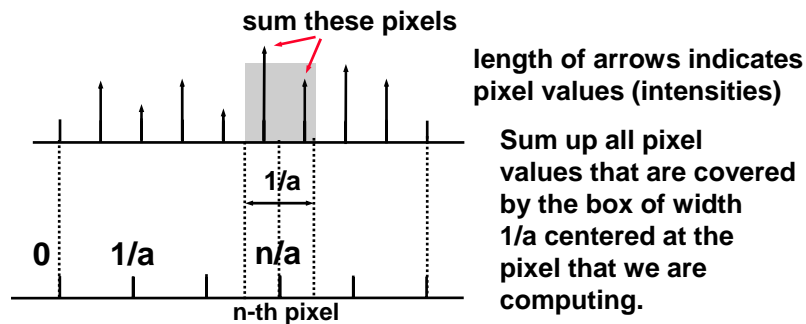
target size, aw



think about pixels of the target as “fat pixels”; the size of a fat pixel is $1/a$; the the size of a the (rescaled) target image is $aw(1/a) = w =$ size of the original (but the pixel size is different!)

© 2001, Denis Zorin

Image shrinking



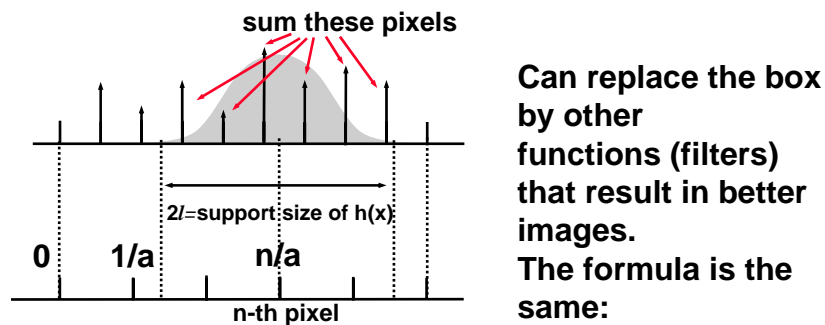
If box function of width 1 is $h(x)$, then:

- box function of width $1/a$ is $h(ax)$
- box function of width $1/a$ centered at n/a is $h(a(x-n/a)) = h(ax-n)$
- add a scale factor a , so that we do not get high intensities:

$$p^{\text{shrunked}}[n] = \sum_i h(ai - n)p[i]$$

© 2001, Denis Zorin

Image shrinking



$$p^{\text{shrunked}}[n] = \sum_i h(ai - n)p[i]$$

We have to sum only over pixels for which $h(ai - n)$ is not zero. If $h(x)$ is zero outside $[-l, l]$, The range for i is determined from

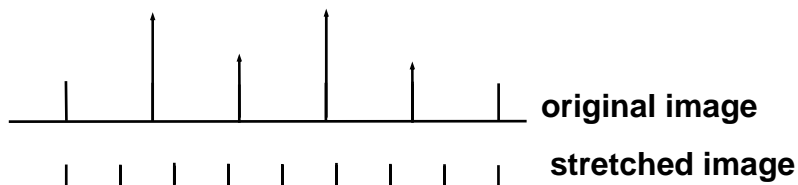
$$-l \leq ai - n \leq l$$

© 2001, Denis Zorin

Image stretching

Stretching by the factor of $a > 1$.

Same approach: make images the same size, use “tiny pixels” of size $1/a$.

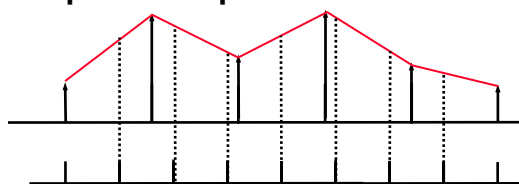


How do we determine values for points between the original pixels? Need to interpolate, that is, find a continuous function coinciding with the original at discrete values.

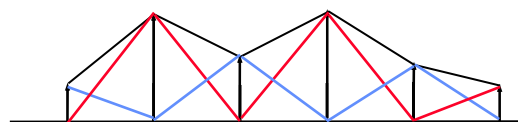
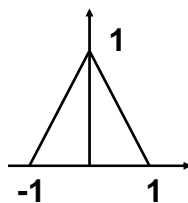
© 2001, Denis Zorin

Interpolation

Simplest interpolation: linear.



How do we write expression for the interpolating function? Use hat functions (one of possible “bumps”).



Put a hat of height $p[n]$ centered at n .
Add all hats.

© 2001, Denis Zorin

Interpolation

Hat functions are linear on integer intervals; when we sum them we get a linear function with values $p[n]$ at integers:

$$f(x) = \sum_i \text{hat}(x - i)p[i]$$

Can use other functions instead of $\text{hat}(x)$:

Just need:

- $h(0) = 1$
- $h(n) = 0$ for n not equal to zero
- interpolating function: $f(x) = \sum_i h(x - i)p[i]$

© 2001, Denis Zorin

Image stretching

Now we only need to resample the interpolating function at “tiny pixel” intervals $1/a$:

$$p[n] = \sum_i h\left(\frac{n}{a} - i\right)p[i]$$

Again, the interval over which to sum is determined by the interval $[-l, l]$, on which the $h(t)$ is not zero:

$$-l \leq \frac{n}{a} - i \leq l$$

© 2001, Denis Zorin

Practical filters

Try to approximate sinc. At the same time, keep short the interval where the filter is not zero.

In addition to box and hat, here is a couple of useful filters:

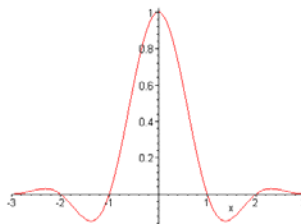
Lanczos filter:
$$h(x) = \begin{cases} \text{sinc}(x)\text{sinc}(x/3), & \text{if } |x| < 3 \\ 0 & \text{otherwise.} \end{cases}$$

Mitchell filter:
$$h(x) = \frac{1}{6} \begin{cases} 7|x|^3 - 12|x|^2 + 16/3, & \text{if } |x| < 1 \\ -7/3|x|^3 + 12|x|^2 - 20|x| + 32/3, & \text{if } 1 \leq |x| < 2 \\ 0 & \text{otherwise} \end{cases}$$

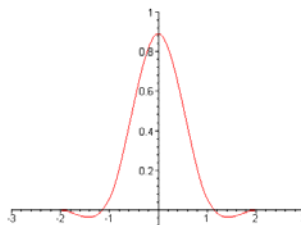
© 2001, Denis Zorin

Practical filters

Lanczos filter, $l=3$:



Mitchell filter, $l=2$:

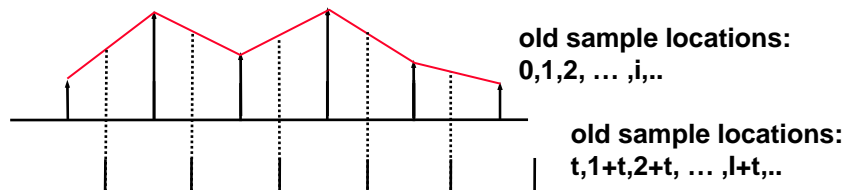


© 2001, Denis Zorin

Image shifts

Similar to image stretching:

Interpolate, then sample at new locations.



$$p^{\text{shifted}}[n] = \sum_i h(n+t-i)p[i]$$

© 2001, Denis Zorin

Implementation summary

To implement resizing (or shifts)

1. create a temporary image. If resizing by factor a in x direction and b in y direction, the temporary image should be $a*w$ by $b*h$, if the original was w by h . For shifts, use the same size.
2. resize/shift in X direction using formulas from lectures, computing pixels in the temporary image using pixels of the original image.
3. Create a final image of size $a*w$ by $b*h$, if resizing, w by h if shifting. Resize/shift in Y direction, computing the pixels of the final image using the pixels of the temporary image.

© 2001, Denis Zorin

Implementation summary

Do all calculations for red, green and blue components of the image separately, that is, new red values are computed from old red values etc.

Represent pixels as **floats**: the results of filtering can be negative or outside 0..255 range; truncate only the final result to this range and convert it to integer.

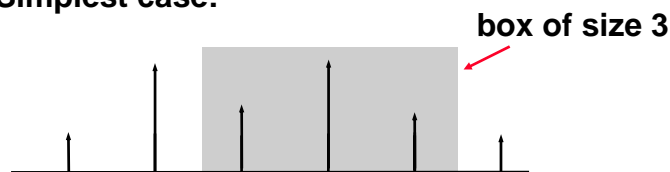
If you need a pixel value that falls outside the image, use zero. Write a function that returns a value for any integer pair of arguments (i,j). If (i,j) is inside the image it returns the image value, otherwise a zero. Always access the image using this function.

© 2001, Denis Zorin

Image blurring

To blur an image, we do local averaging.

Simplest case:



new pixel value here is the average of the three old pixel values.

$$p^{\text{filtered}}[n] = \sum_i h(n-i)p[i]$$

Note: we need values of $h(t)$ only at integers.

© 2001, Denis Zorin

Discrete filtering

When we do not do resampling at arbitrary locations, as we do when resizing the image, we can use discrete filters $h[i]$, which are just sequences of numbers. One way to obtain such filters is to sample a continuous filter.

Aside from blurring, other effects can be achieved using discrete filters: e.g. edge detection and sharpening.

© 2001, Denis Zorin

Discrete filtering

Convolution: Given two discrete sequences $p[i]$ and $h[i]$, $i = -\infty.. \infty$, the convolution of these sequences is a new sequence $q[i]$ defined by

$$q[n] = \sum_{i=-\infty}^{\infty} h[n-i]p[i]$$

Discrete filtering is convolution of a signal with a filter (Of course, the summation is not really infinite as both sequences are finite; they are assumed to be extended to both sides by zeros when necessary).

© 2001, Denis Zorin

2D convolution

2D convolution:

$$q[n,m] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} h[n-i, m-j] p[i, j]$$

Note: not for any $h[i,j]$ we can implement 2D convolution as a sequence of 2 1D convolutions. Convolution is implemented as 4 nested loops.

Typically, in formulas the indices of filters run in both directions from $-L$ to L , where L is an integer. Be careful when retrieving the filter values from an array: you have to convert the range $[-L..L]$ to $[0..2*L]$

© 2001, Denis Zorin

Edge detection

Idea: an edge is a sharp change in the image. To find edges means to mark with, say, 255, all pixels which are on an edge. For a continuous image, the places where the intensity changes rapidly, the magnitude of the derivative in some direction is large.

Directional derivatives can be approximated by differences:

$$\frac{df(x,y)}{dx} \approx \frac{f(x+1,y) - f(x,y)}{1} = f(x+1,y) - f(x,y)$$

Differences can be computed using filtering.

© 2001, Denis Zorin

Difference filters

To compute

$$\Delta_x p[n,m] = p[n+1,m] - p[n,m]$$

convolve with filter $h[0,0] = -1$, $h[-1,0] = 1$,
 $h[i,j] = 0$ otherwise.

To compute

$$\Delta_y p[n,m] = p[n,m+1] - p[n,m]$$

convolve with filter $h[0,0] = -1$, $h[0,-1] = 1$,
 $h[i,j] = 0$ otherwise.

© 2001, Denis Zorin

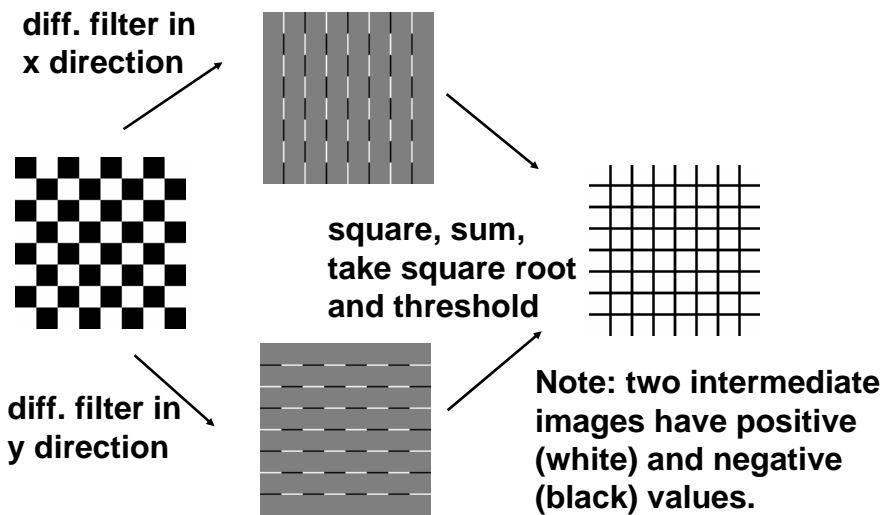
Edge detection

To mark locations where the differences are large,
compute differences in two directions, square, and
threshold:

if $\sqrt{\Delta_x p[n,m]^2 + \Delta_y p[n,m]^2} > \text{threshold_value}$
set $p^{\text{edge}}[n,m]$ to 255, otherwise, to zero.

© 2001, Denis Zorin

Edge detection example



© 2001, Denis Zorin

Sharpening

Idea: to sharpen an image, that is, to make edges more apparent, need to increase the high frequency component and decrease low frequency component. Can be achieved by subtracting a scaled blurred version of the image from the original.

The operation can be done using a single 2D convolution by a filter like this:

$$\frac{1}{7} \begin{bmatrix} -1 & -2 & -1 \\ -2 & 19 & -2 \\ -1 & -2 & -1 \end{bmatrix}$$

© 2001, Denis Zorin