Application of Brent's Theorem to our Optimization of Prefix-Sum

- Assume that the optimized version loads f floats into local registers
- Work complexity:
  - Without optimization: $W_1(n) = 2n$
  - With optimization: $W_2(n) = 2\frac{n}{f} + \frac{n}{f} \cdot f = n(1 + \frac{2}{f})$
- Depth complexity:
  - Without optimization: $D_1(n) = 2\log(n)$
  - With optimization: $D_2(n) = 2\log(\frac{n}{f}) + f = 2\log n - 2\log f + f$
- If $f = 2$, then $W_2 = W_1$ and $D_2 = D_1$, i.e., we gain nothing
- If $f > 2$, speedup of version 2 (opt.) over version 1 (original):

$$\text{Speedup}(n) = \frac{T_2(n)}{T_1(n)} = \frac{\frac{W_1(n)}{p} + D_1(n)}{\frac{W_2(n)}{p} + D_2(n)} \approx \frac{2\frac{n}{p}}{\frac{n}{p}(1 + \frac{2}{f})} = \frac{2f}{f+2}$$

Other Consequences of Brent's Theorem

- Obviously, $\text{Speedup}(n) \leq p$
- In the sequential world, time = work: $T_S(n) = W_S(n)$
- In the parallel world: $T_P(n) = \frac{W_P(n)}{p} + D(n)$
- Our speedup is $\text{Speedup}(n) = \frac{T_S(n)}{T_P(n)} = \frac{W_S(n)}{\frac{W_P(n)}{p} + D(n)}$
- Assume, $W_P(n) \in \Omega(\, W_S(n)\,)$

  i.e., our parallel algorithm would do asymptotically more work
- Then, $\text{Speedup}(n) = \dfrac{W_S(n)}{\Omega(\, W_S(n)\,) + D(n)} \rightarrow 0 \quad \text{as } n \rightarrow \infty$

  because, on real hardware, p is bounded
- This is the reason why we want work-efficient parallel algorithms!

- Now, look at work-efficient parallel algorithms, i.e.

$$W_P(n) \in \Theta(\, W_S(n)\,)$$

- Then,

$$\text{Speedup}(n) = \frac{W(n)}{\frac{W(n)}{p} + D(n)} = \frac{pW(n)}{W(n) + pD(n)}$$

- In this situation, we will achieve the optimal speedup of p, so long as

$$p \in O(\frac{W(n)}{D(n)})$$

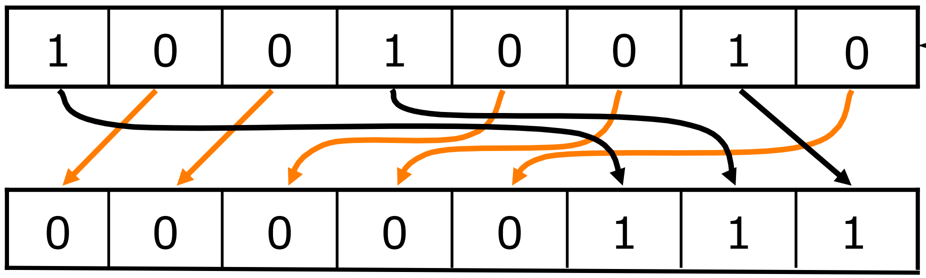
- Consequence: given two work-efficient parallel algorithms, the one with the smaller depth complexity is better, because we can run it on hardware with more processors (cores) and still obtain a speedup of p over the sequential algorithm (in theory). We say this algorithm scales better.

Limitations of Brent's Theorem

- Brent's theorem is based on the PRAM model
- That model makes a number of unrealistic assumption:
  - Memory access has zero latency
  - Memory bandwidth is infinite
  - No synchronization among processors (threads) is necessary
  - Arithmetic operations cost unit time
- With current hardware, rather the opposite is realistic

Radix Sort, Based on the Split Operation

- The split operation: rearrange elements according to a flag



Flags.
There could be
payload data, too
(omitted here)

  - Note: split maintains order within each group! (i.e., it is stable)

- Radix sort (massively parallel):

```
radix_sort( array a, int len ):
  for i = 0...numbits-1:  // important: go from low to high bit!
    split(i, a)           // split a, based on bit i of keys
```

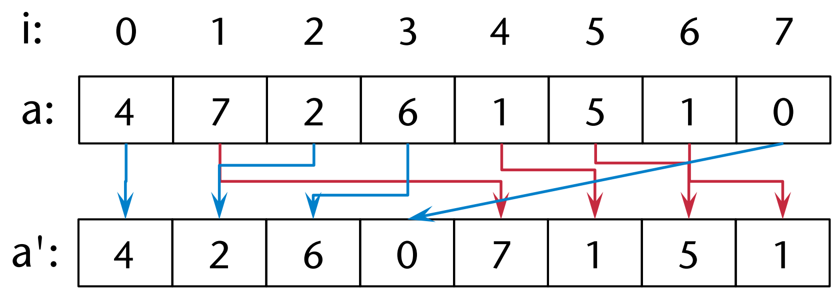
where **split(i,a)** rearranges **a** by moving all keys that have bit **i** = 0 to the bottom, all keys that have bit **i** = 1 to the top (lowest bit = bit no. 0)

  - Reminder: stability of *split* is essential!

Algorithm for the Split Operation

- Split's job:
  - Determine new index for each element
  - Then perform the permutation
- Algorithm (by way of an example):
  - Consider lowest bit of the keys

1. Compute "0"-scan (exclusive):
   f_i = # "0"s in $(a_0, \ldots, a_{i-1})$
2. Set F = total number of "0"s
   $$= \begin{cases} f_{n-1} + 1 & a_{n-1} = 0 \\ f_{n-1} & a_{n-1} = 1 \end{cases}$$
3. If $a_i = 0 \rightarrow$ new pos. $d = f_i$
4. If $a_i = 1 \rightarrow$ new pos. $d = F + (i - f_i)$
   - Because $i - f_i$ = # "1"s to the left of i



i:	0	1	2	3	4	5	6	7
a:	100	111	010	110	001	101	001	000
f:	0	1	1	2	3	3	3	3
F=4								
$d_{\text{for "0"s}}$:	0		1	2				3
$d_{\text{for "1"s}}$:		4+(1-1)			4+(4-3)	4+(5-3)	4+(6-3)	
d:	0	4	1	2	5	6	7	3

- A conceptual algorithm for the "0"-scan:
  - Extract the relevant bit (conceptually only)
  - Invert the bit
  - Compute regular scan with +-operation

a:	100	111	010	110	001	101	001	000
a':	1	0	1	1	0	0	0	1
f:	0	1	1	2	3	3	3	3

- In a real implementation, you would, of course, implement this as a native "0"-scan routine!

Stream Compaction

- Given: input stream A, and a *flag/predicate* for each a_i
- Goal: output stream A' that contains only a_i's, for which flag = true
- Example:
  - Given: array of upper and lower case letters
  - Goal: delete lower case letters and compact the upper case to the low-order end of the array

a:	A	X	C	P	H	W	B	Z
a':	1	0	1	1	0	0	0	1
b:	A	C	P	Z				

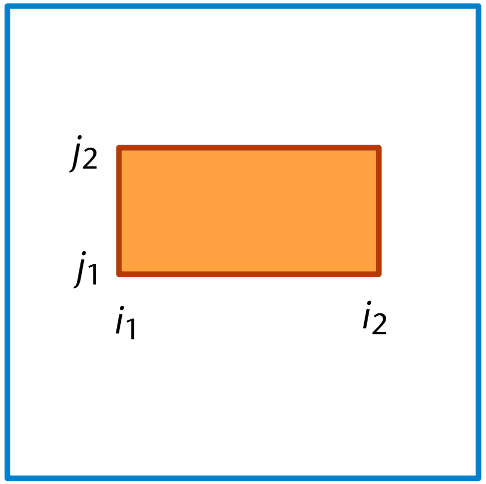
- Solution:
  - Just like with the split operation, except we don't compute indices for the "false" elements
- Frequent task: e.g., collision detection,
- Sometimes also called list packing, or stream packing

Summed-Area Tables / Integral Images

- Given: 2D array T of size $w \times h$
- Wanted: a data structure that allows to compute

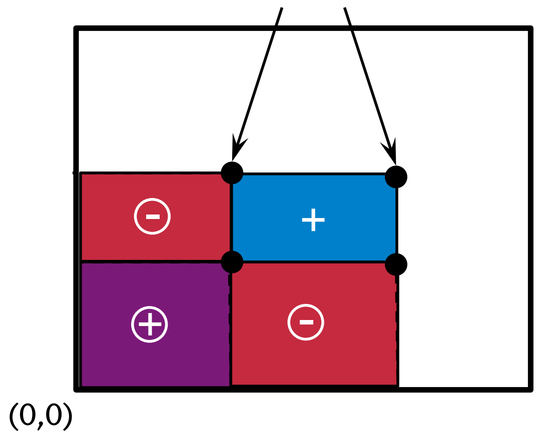
$$\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k,l)$$

for any i_1, i_2, j_1, j_2 in $O(1)$ time



- The trick:

$$\sum_{k=i_1}^{i_2}\sum_{l=j_1}^{j_2} T(k,l) = \sum_{k=1}^{i_2}\sum_{l=1}^{j_2} T(k,l) - \sum_{k=1}^{i_1}\sum_{l=1}^{j_2} T(k,l) - \sum_{k=1}^{i_2}\sum_{l=1}^{j_1} T(k,l) + \sum_{k=1}^{i_1}\sum_{l=1}^{j_1} T(k,l)$$



- Define

$$S(i,j) = \sum_{k=1}^{i}\sum_{l=1}^{j} T(k,l)$$

- With that, we can rewrite the sum:

$$\sum_{k=i_1}^{i_2}\sum_{l=j_1}^{j_2} T(k,l) = S(i_2,j_2) - S(i_1,j_2) - S(i_2,j_1) + S(i_1,j_1)$$

- Definition:
  Given a 2D (k-D) array of numbers, T, the summed area table S stores for each index (i,j) the sum of all elements in the rectangle (0,0) and (i,j) (inclusively):

$$S(i,j) = \sum_{k=1}^{i} \sum_{l=1}^{j} T(k,l)$$

- Like prefix-sum, but for higher dimensions
- In computer vision, it is often called integral image
- Example:

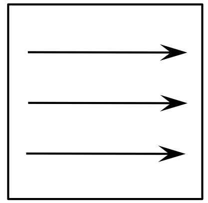
Input

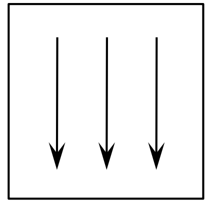
2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

Summed Area Table

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

- The algorithm: 2 phases (for 2D)
  1. Do H prefix-sums horizontally
  2. Do W prefix-sums vertically
     - Real implementation (to maintain *coalesced memory access*): prefix-sum vertically, transpose, prefix-sum vertically
     - Or use texture memory





- Depth complexity for k-D (assume $w = h$, and "native" horizontal prefix-sum, i.e., no transposition):

$$k \cdot W \log W$$

- Caveat: precision of integer/floating-point arithmetic
  - Assumption: each T_{ij} needs b bits
  - Consequence: number of bits needed for $S_{wh} = \log w + \log h + b$
  - Example: 1024x1024 grey scale input image, each pixel = 8 bits
    → 28 bits needed in S-pixels

Increasing the Precision

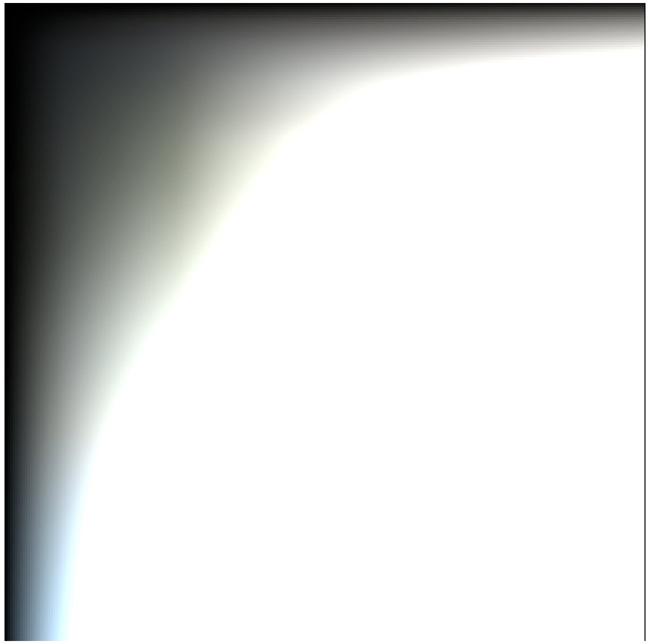
- The following techniques actually apply to prefix-sums, too!

1. "Signed offset" representation:
   - Set
     $$T'(i,j) = T(i,j) - \bar{t}$$
     where $\bar{t} =$ average of $T = \frac{1}{wh} \sum_1^w \sum_1^h T(i,j)$
   - Effectively removes DC component from signal
   - Consequence:
     $$S'(i,j) = \sum_{k=1}^{i} \sum_{l=1}^{j} T'(k,l) = S(i,j) - i \cdot j \cdot \bar{t}$$
     i.e., the values of S' are now in the same order as the values of T (less bits have to be thrown away during the summation)
   - Note 1: we need to set aside 1 bit (sign bit)
   - Note 2: $S'(w,h) = 0$ (modulo rounding errors)

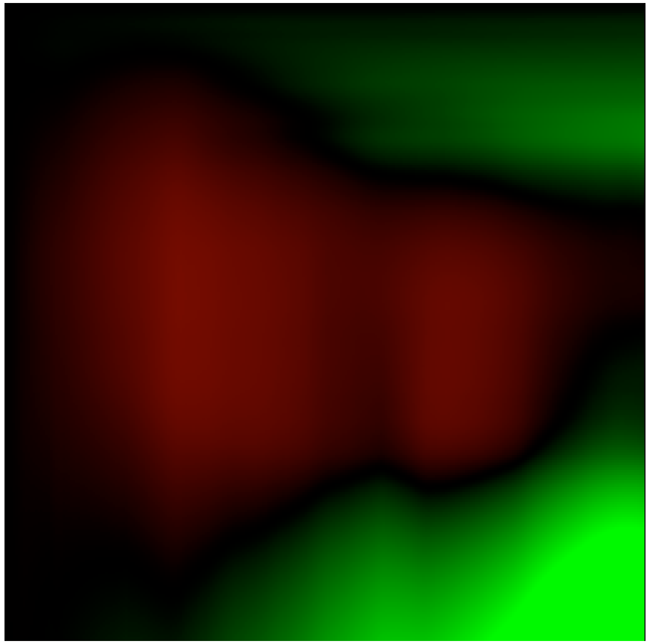
- Example:

Input image

Original summed area table

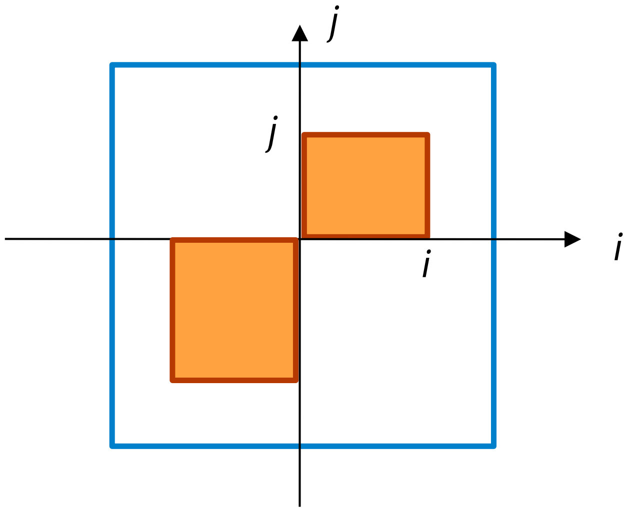


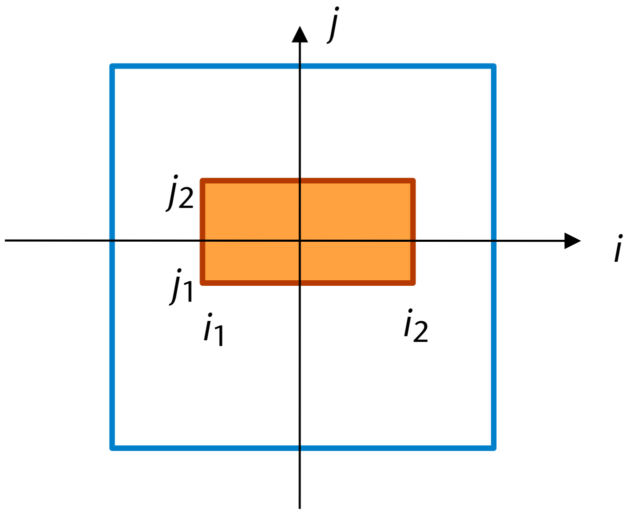
Improved precision
using "offset" representation



2. Move the "origin" of the *i*,*j* "coordinate frame":
   - Compute 4 different *S*-tables, one for each quadrant
   - Result: each *S*-table comprises only ¼ of the pixels/values of *T*

- For computation of $\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k,l)$ do a simple case switch





Results

- Compute integral image
- From that, compute

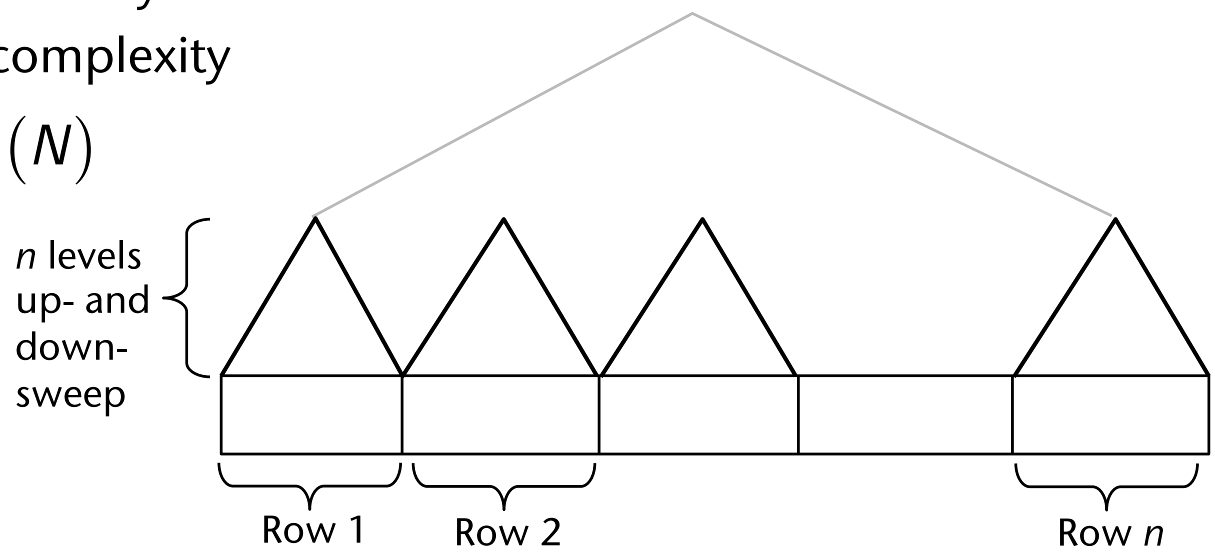
$$
\begin{aligned}
&S(i,j)\\
-&S(i-1,j)\\
-&S(i,j-1)\\
+&S(i-1,j-1)
\end{aligned}
$$

  - I.e., 1-pixel box filter
- Should yield the original image (theoretically)

With methods 1 & 2 | Simple method

Efficient Computation of the Integral Image

- Naïve approach: do a 1D prefix-sum per row → $O\left(\sqrt{N}\,\log N\right)$ depth complexity (assuming we omit the matrix transposition step) and $O\left(\sqrt{N}\cdot\sqrt{N}\right) = O(N)$ work complexity, where input image has size $n \times n = N$ pixels
- Better solution:
  - Pack all rows into one linear array of size N
  - Do a 1D prefix-sum, but only the first n levels
    → $O(\log N)$ depth complexity
  - Work complexity = $O(N)$
- Is a special case of segmented prefix sum

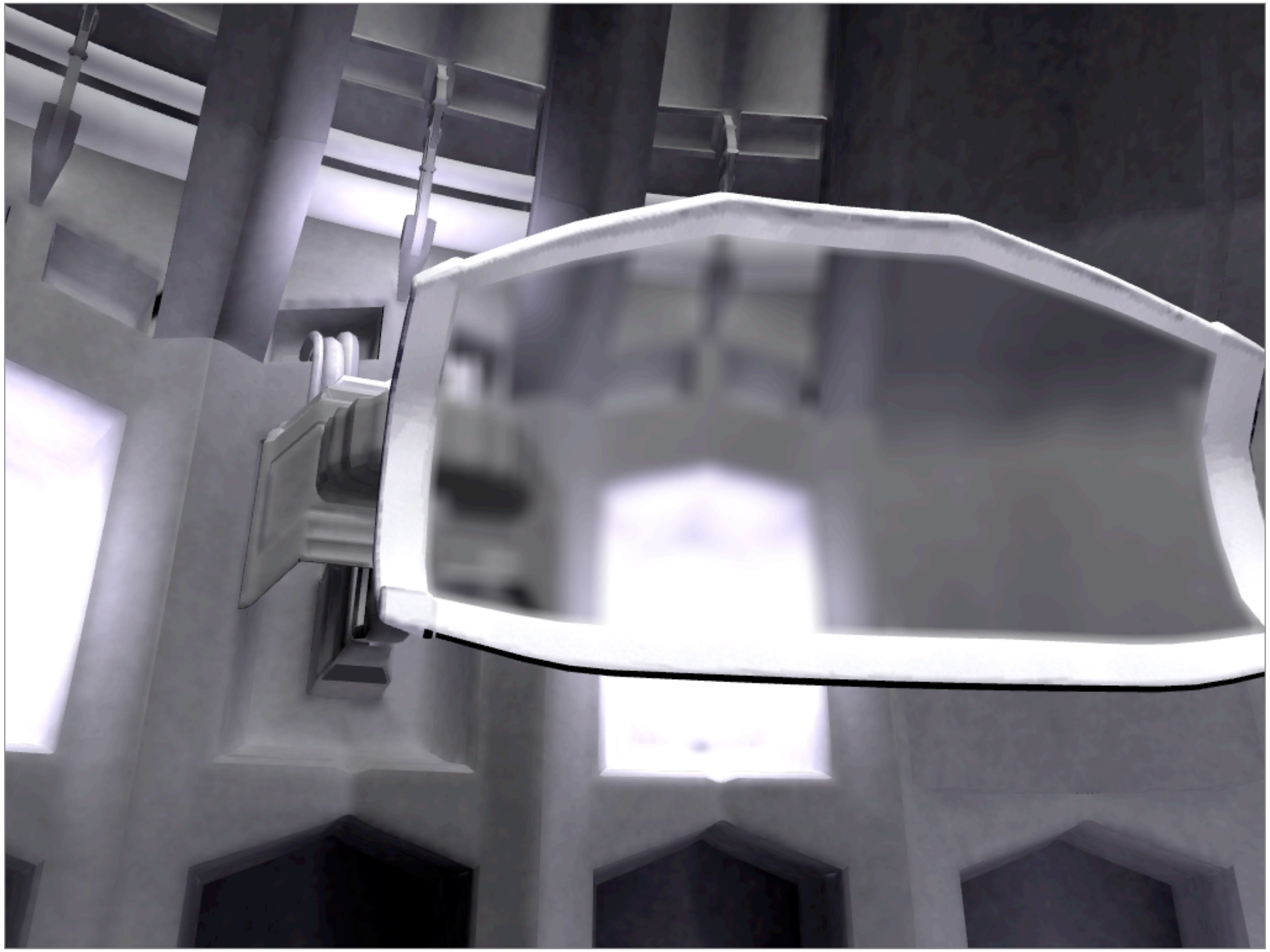


Applications of the Summed Area Table

- For filtering in general
- Simple example: box filter
  - Compute average inside a box (= rectangle)
  - Slide box across image (convolution)
- Application: translucent objects, i.e., transparent & matte
  - E.g., milky glass

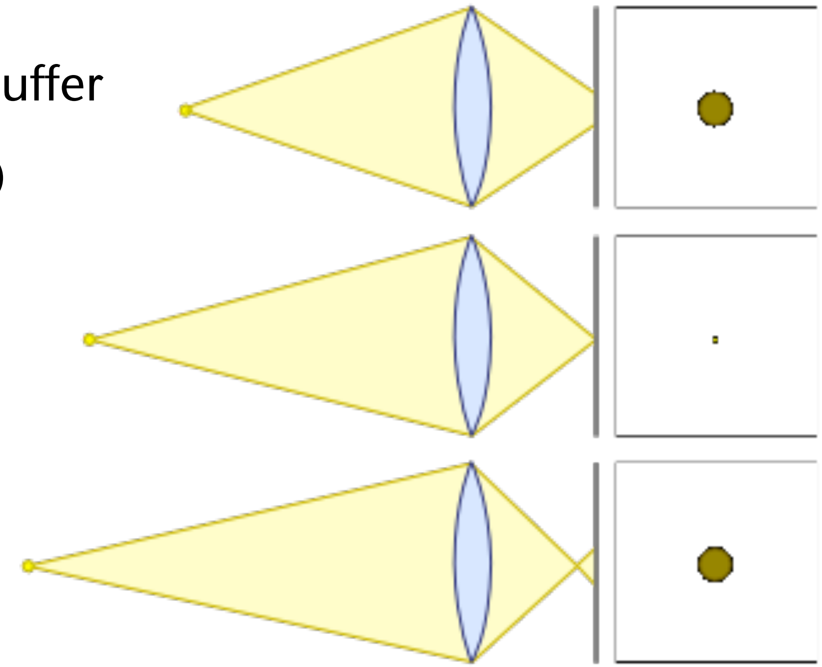
1. Render virtual scene (e.g., game) without translucent objects
2. Compute summed area table from frame buffer
3. Render translucent object (using fragment shader): replace pixel behind translucent object by average over original image within a (small) box

- Result:



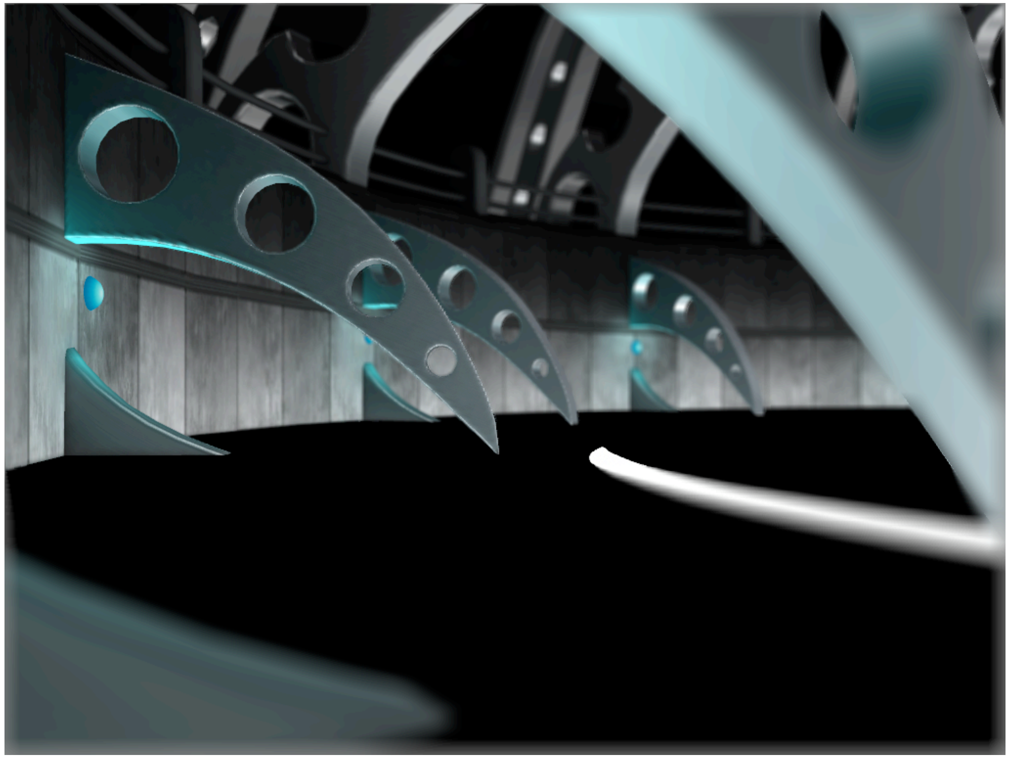
Rendering with Depth-of-Field (Tiefenunschärfe)

1. Render scene, save color buffer and z-buffer (e.g., in texture)
2. Compute summed area table over color buffer
3. For each pixel do *in parallel*:
   1. Read depth of pixel from saved z-buffer
   2. Compute circle of confusion (CoC) (for details see "Advanced CG")
   3. Determine size of box filter
   4. Compute average over saved color buffer within box
   5. Write in color buffer



- Note: "For each pixel in parallel" could be implemented in OpenGL by rendering a screen-filling quad using special fragment shader

- Result:

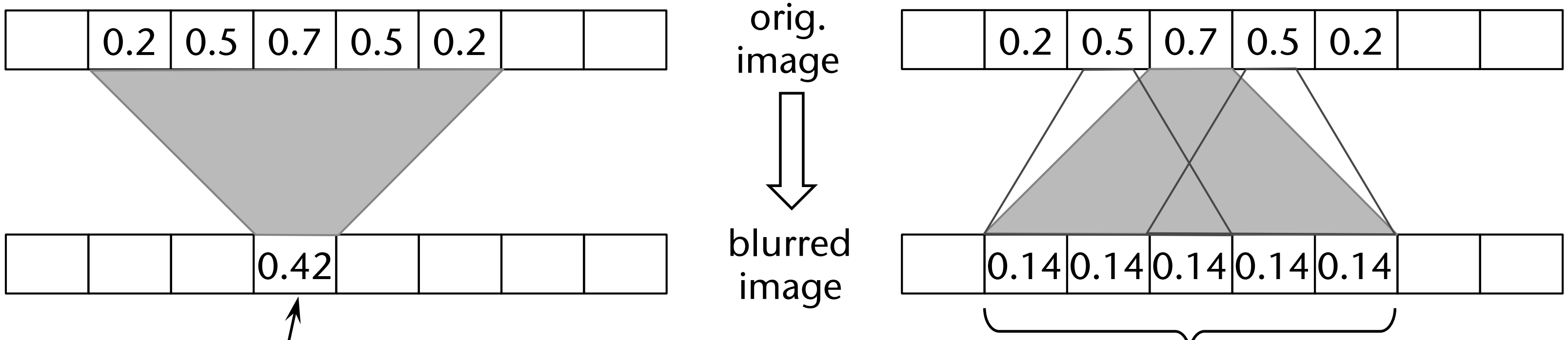


Artifacts of this Technique

- False sharp silhouettes: blurry objects (out of focus) have sharp silhouette, i.e., won't blur over sharp object (in focus)
- Color bleeding (a.k.a. pixel bleeding): areas in focus can incorrectly bleed into nearby areas out of focus
- Reason: the (indiscriminate) gather operation

Depth-of-Field with Scattering

- Goal: turn gather operation into scatter operation



average gathered over CoC

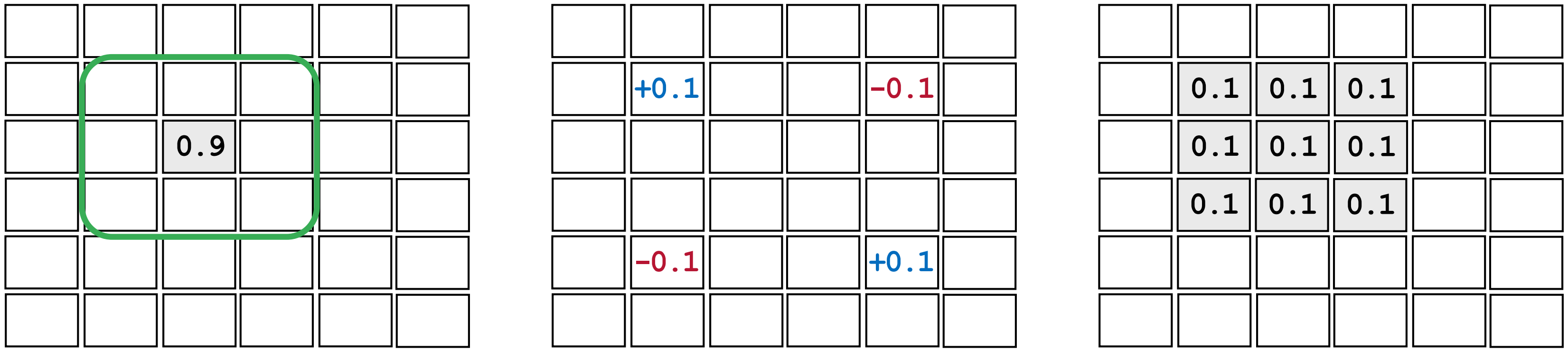
one pixel scattered over CoC

- Example: scatter *one* pixel using the 2D prefix-sum (integral image)

Input image with one pixel set and its "circle"-of-confusion

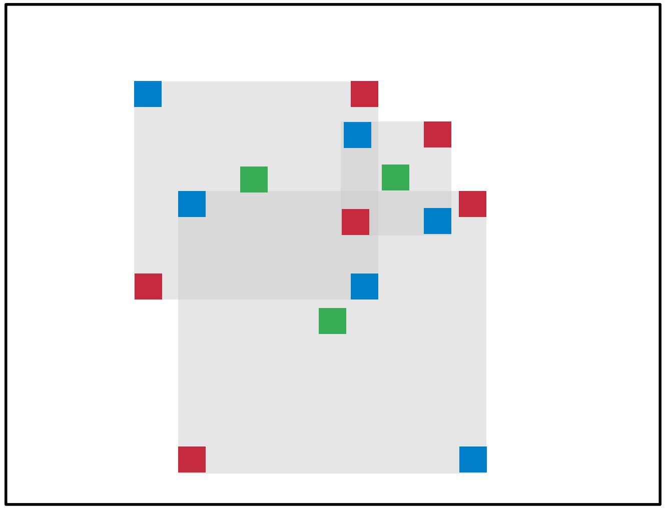
Pixel value spread to the corners of the rectangle

Resulting 2D prefix-sum = pixel scattered over CoC



Algorithm

1. Phase: for each pixel in original image do in parallel
   - Spread $\frac{\text{pixel value}}{\text{area}(\text{CoC})}$ to CoC corners
     - Use atomic accumulation operation !
     - Do this for each R, G, and B channel
2. Phase: compute 2D prefix-sum,
   result = blurred image



- Question: can you turn phase 1 into a *gather phase*?

Result

Summed area table and gathering

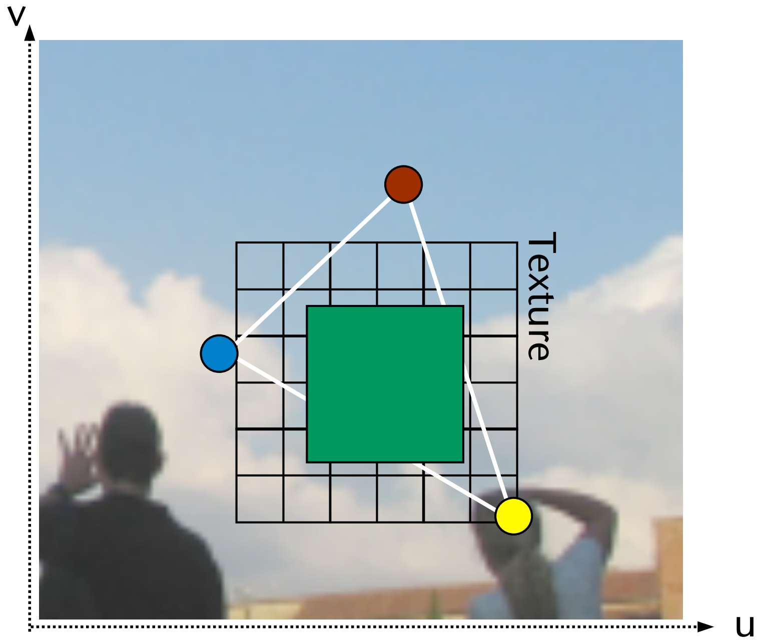
Scattering and 2D prefix-sum

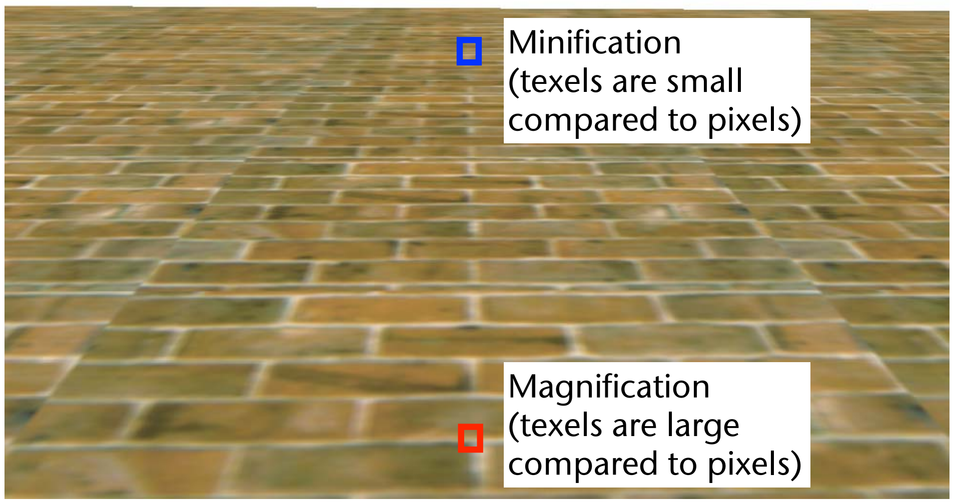
[Kosloff, Tao, Barsky, 2009]

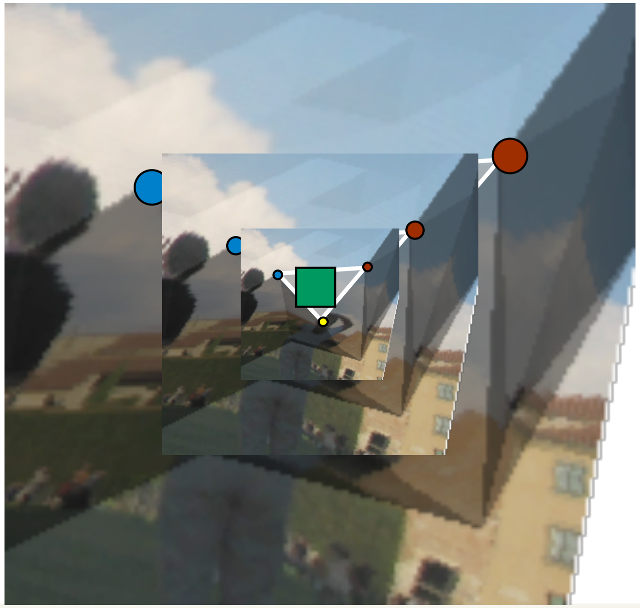
Optional

Recap: Texture Filtering in Case of Minification

- What happens, when we "zoom away" from the polygon?

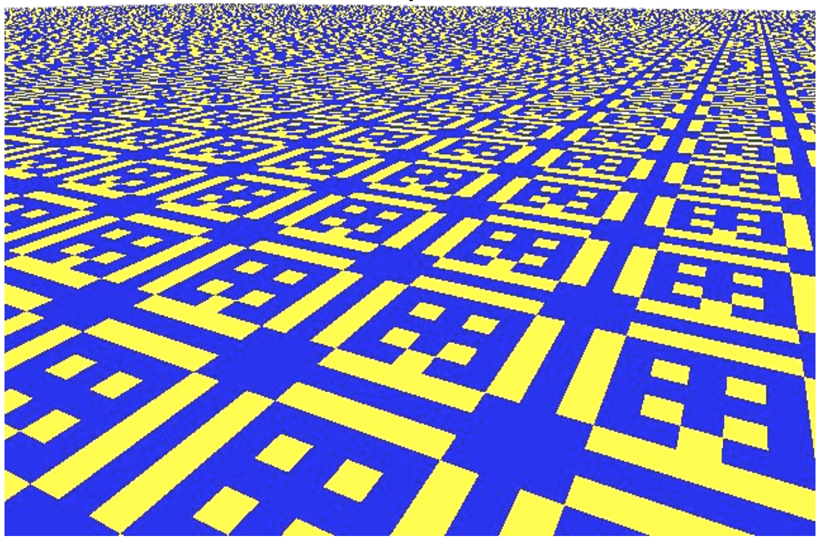




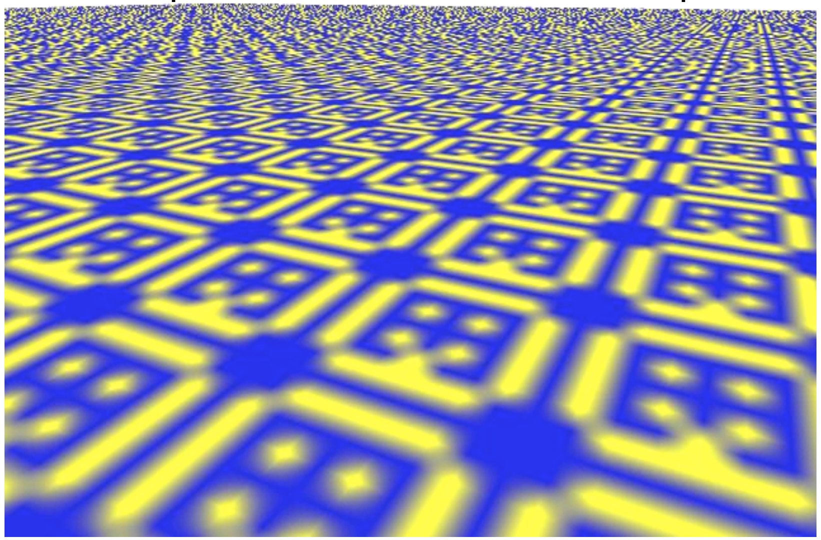


- Linear interpolation does not help very much:

Take texel closest to pixel center (in u,v)

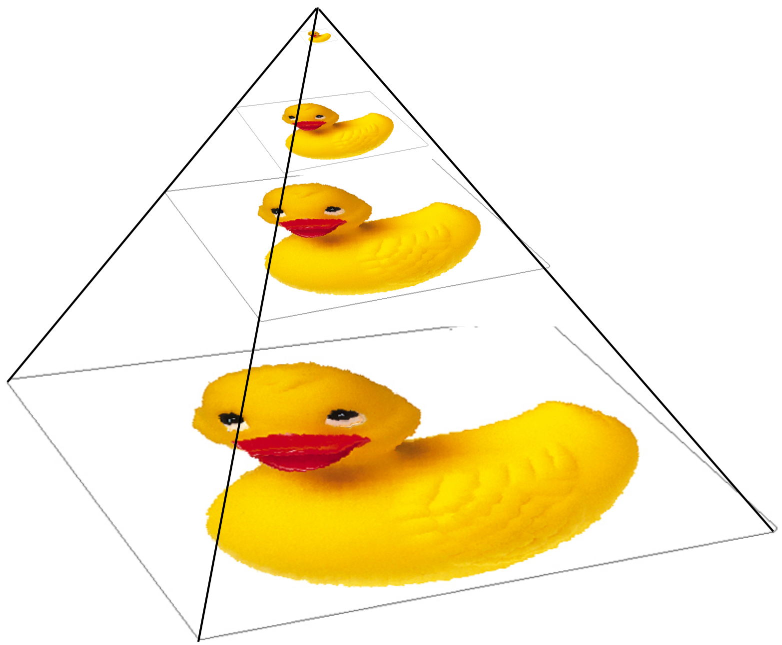


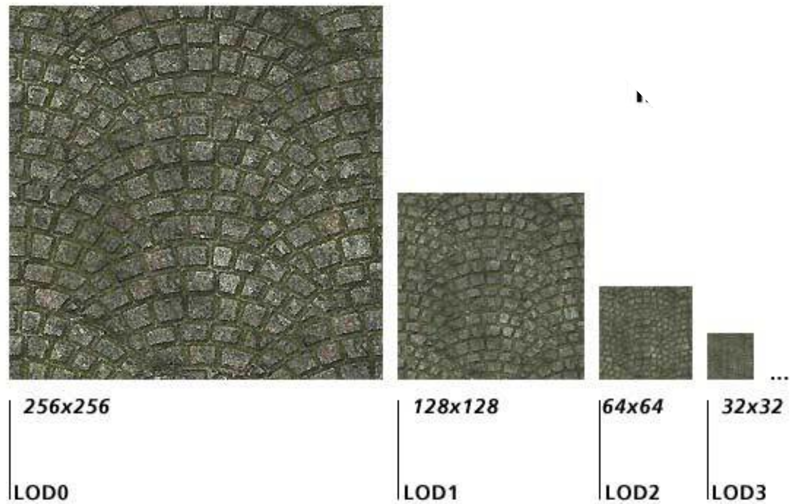
Linear interpolation of 4 texels closest to pixel ctr



- Needed would be an averaging of all texels covered by the pixel (in uv-space); too costly in real-time
- Solution: pre-processing → MIP-Maps
  (lat. "multum in parvo" = Vieles im Kleinen")

- A MIP-Map is just an image pyramid:
  - Each level is obtained by averaging 2x2 pixels of the level below
    - Consequence: the original image must have size $2^n x 2^n$ (at least, in practice)
  - You can use more sophisticated ways of filtering, e.g., Gaussian
- Memory usage for MIP-Map: 1.3x original size

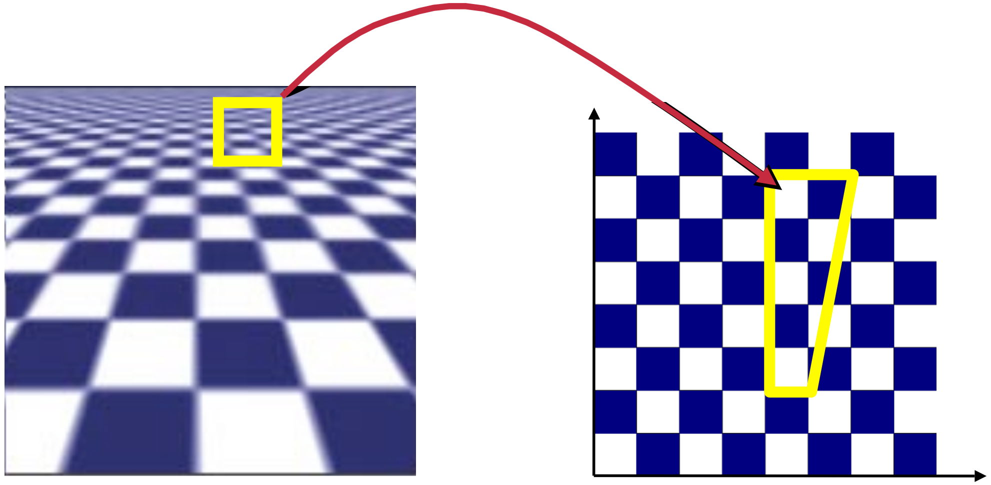


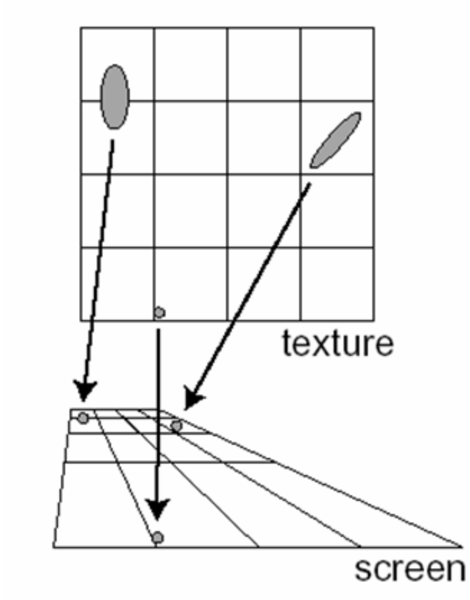


Optional

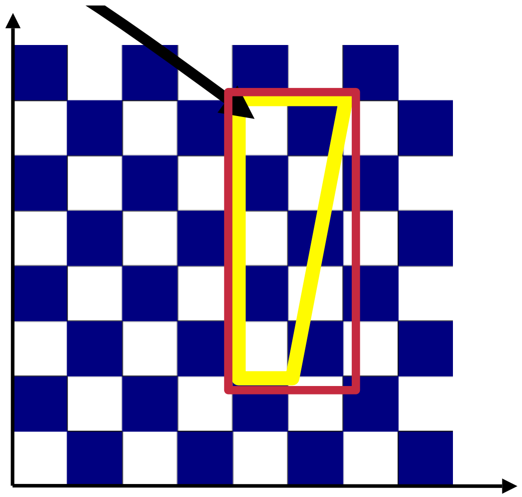
Anisotropic Texture Filtering

- Problem with MIPmapping: doesn't take the "shape" of the pixel in texture space into account!



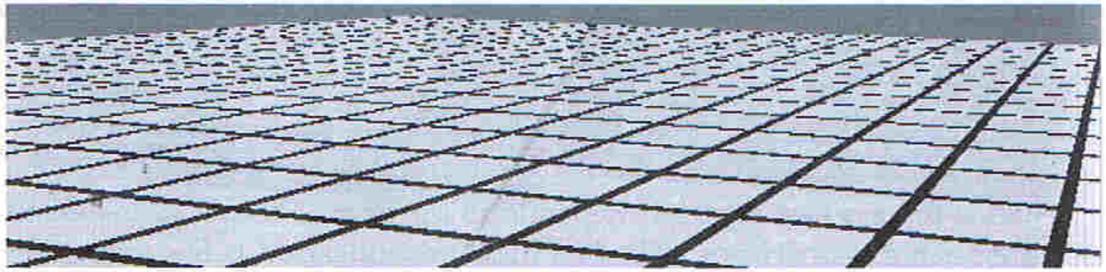


  - MIPmapping just puts a square box around the pixel in texture space and averages all texels within
- Solution: average over bounding rectangle
  - Use Summed Area Table for quick summation
- Question: how to average over highly "oblique" pixels?

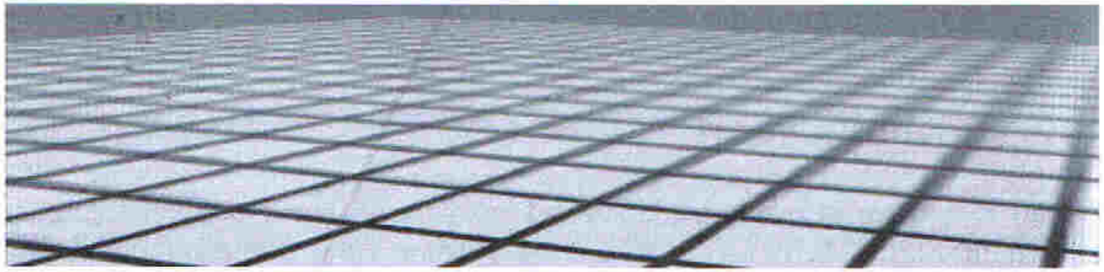


Optional

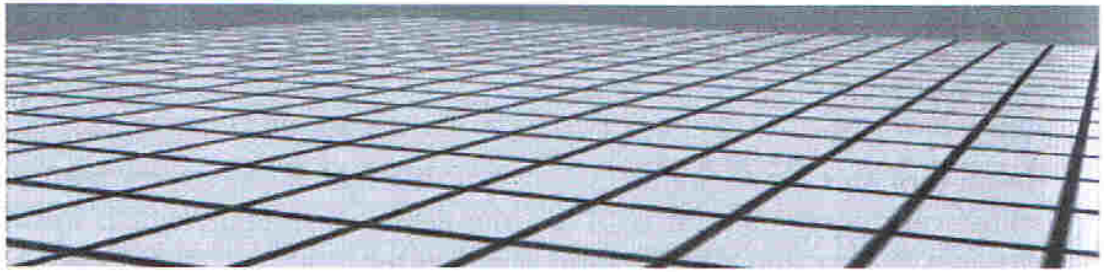
- This is one kind of *anisotropic texture filtering*
- Result:



No filtering



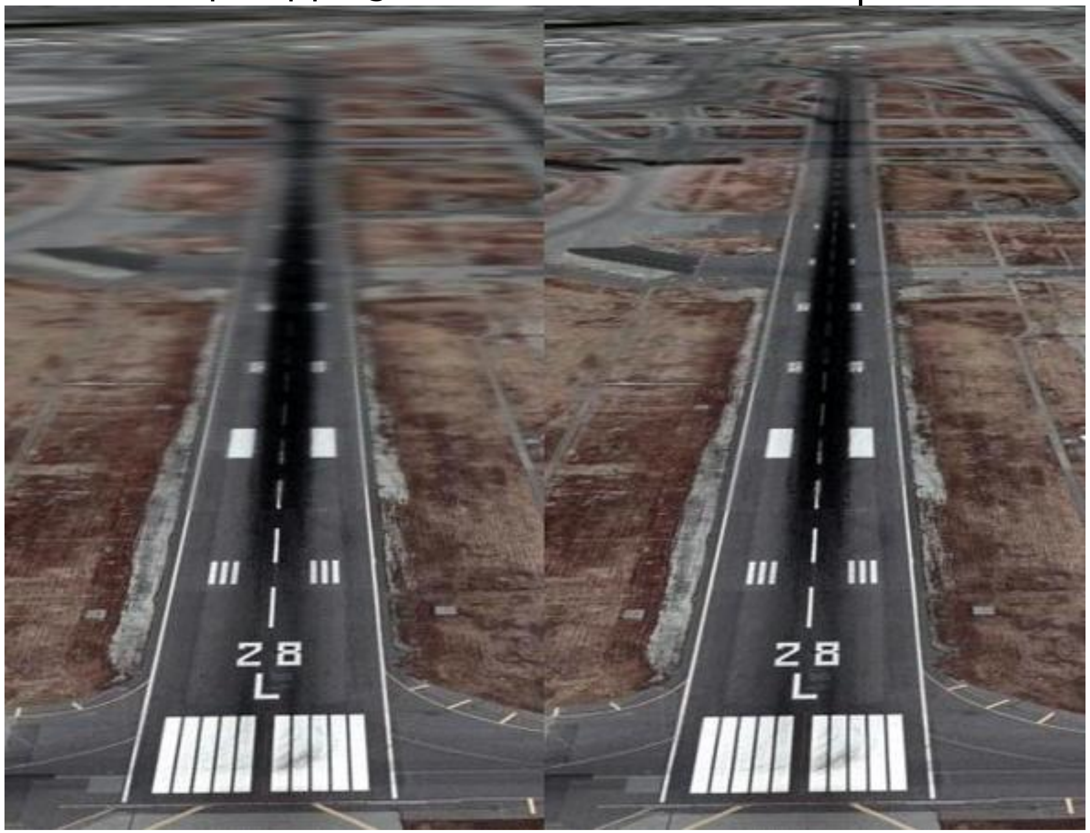
Mipmapping



Summed area table

- Another example:

Mipmapping Anisotropic

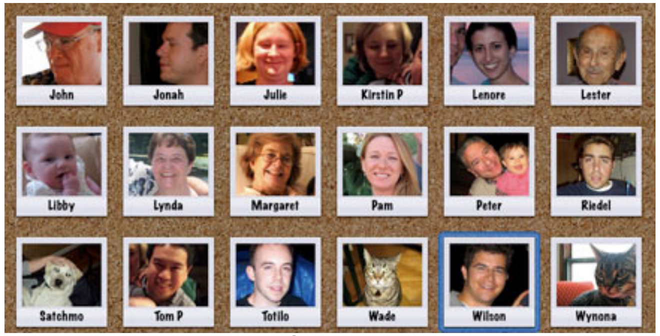


- Today: all graphics cards support anisotropic filtering (not necessarily using SATs)

Application: Face Detection

- Goal: detect faces in images

digital camera



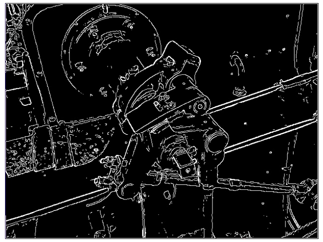
iPhoto

"False positive" from human point of view

- Requirements (wishes):
  - Real-time or close (> 2 frames/sec)
  - Robust (high true-positive rate, low false-positive rate)
- Non-goal: face recognition

- In the following: no details, just overview!

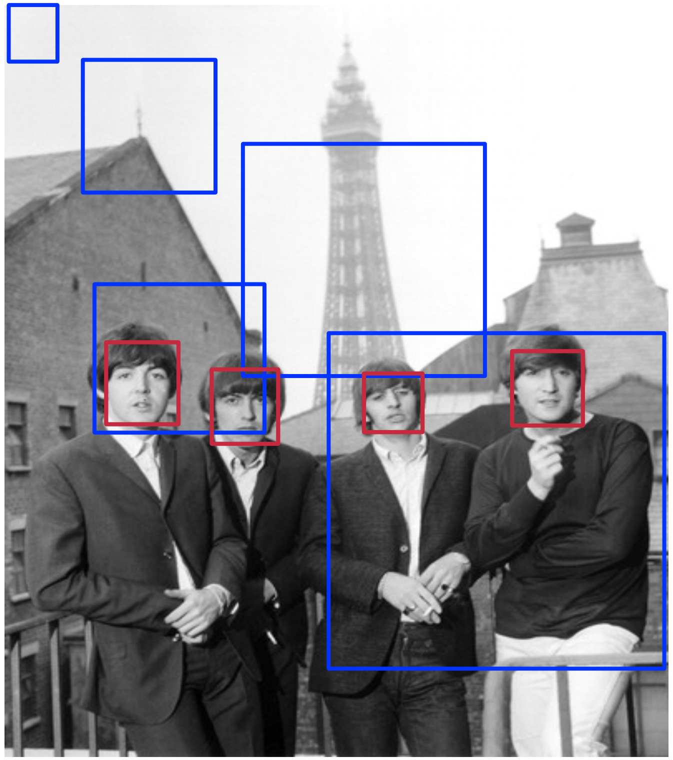
- The term feature in computer vision:
  - Can be literally *any* piece of information/structure present in an image (somehow)
  - *Binary features* → present / not present; examples:
    - Edges (e.g., gradient > threshold)
    - Color of pixels is within specific range (e.g., skin)
    - Ellipse filled with certain amount of skin color pixels
  - *Non-binary features* → probability of occurrence; examples:
    - Gradient image
    - Sum of pixel values within a shape, e.g., rectangle



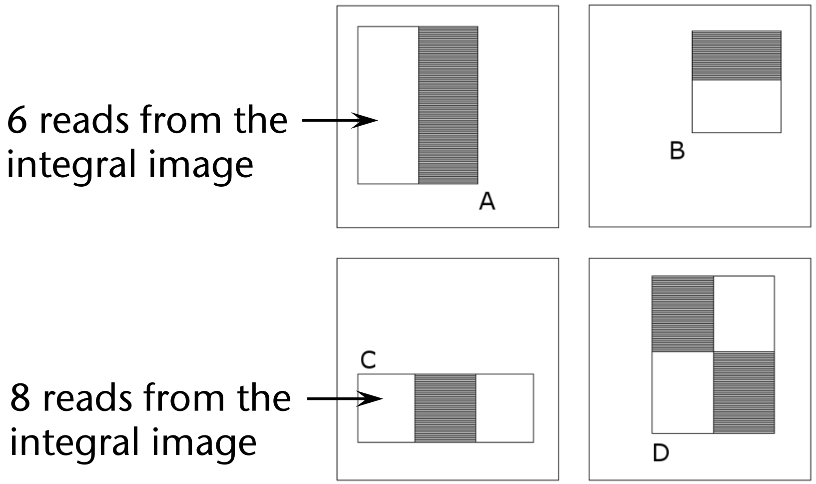
The Viola-Jones Face Detector

[2002]

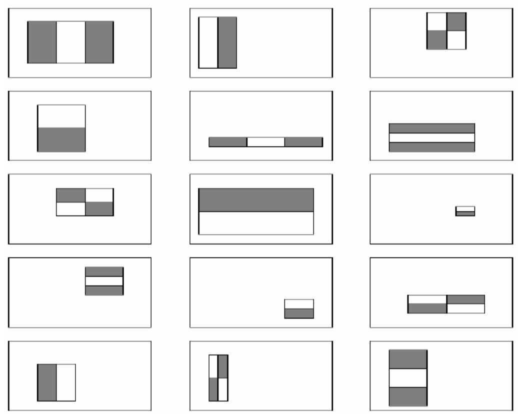
- The (simple) idea:
  - Move sliding window across image (all possible locations, all possible sizes)
  - Check, whether a face is in the window
  - We are interested only in windows that are filled by a face
- Observation:
  - Image contains 10's of faces
  - But ~10^6 candidate windows
- Consequence:
  - To avoid having a false positive in every image, our false positive rate has to be < 10^{-6}



- Feature types used in the Viola-Jones face detector:
  - 2, 3, or 4 rectangles placed next to each other
  - Called Haar features



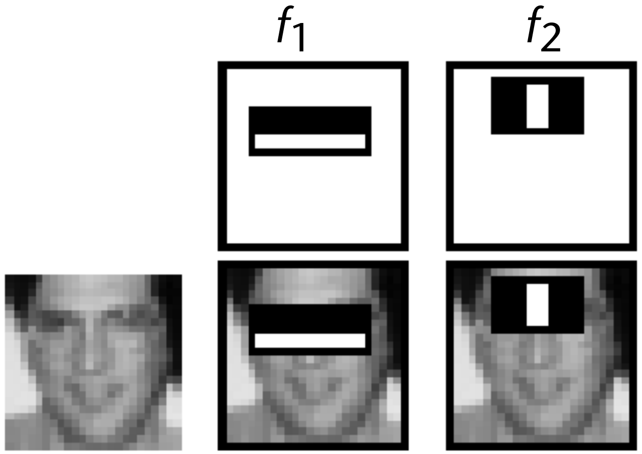
- Feature value := g_i = pixel-sum(white rectangle(s)) – pixel-sum(black rectangle(s))
  - Constant time per feature extraction
- In a 24x24 window, there are ~160,000 possible features
  - All variations of type, size, location within window



- Define a weak classifier for each feature:

$$f_i = \begin{cases} +1 & , g_i > \theta_i \\ -1 & , \text{else} \end{cases}$$

  - "Weak" because such a classifier is only slightly better than a random "classifier"



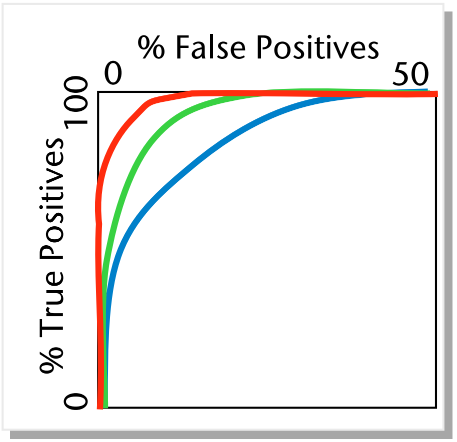
- Goal: combine lots of weak classifiers to form one strong classifier

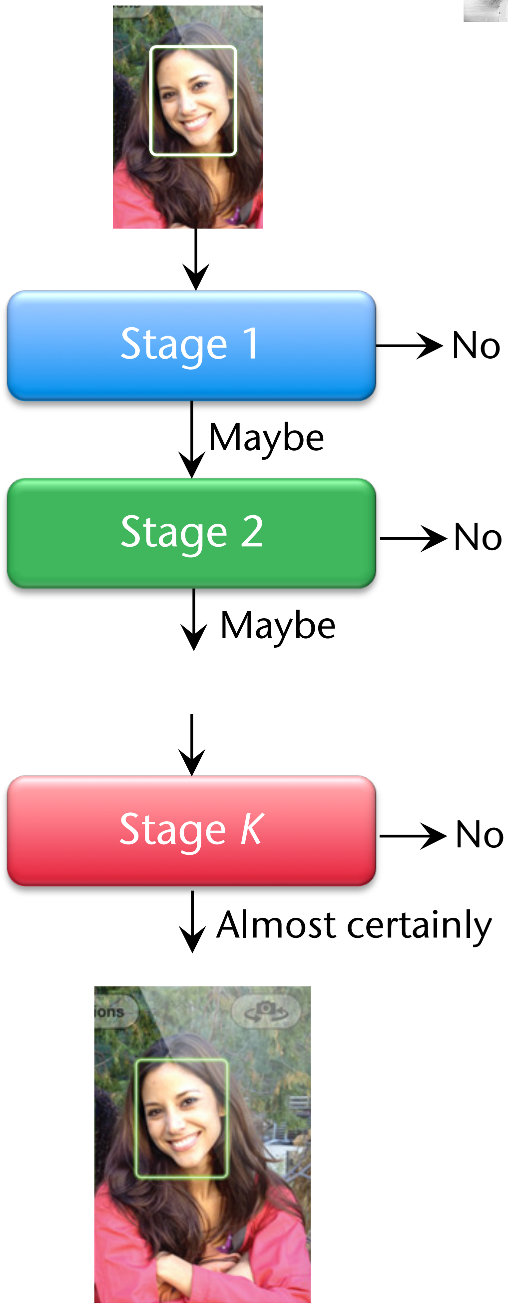
$$F(\text{window}) = \alpha_1 f_1 + \alpha_2 f_2 + \ldots$$

- Use learning algorithms to automatically find a set of *weak classifiers* and their optimal weights and thresholds, which together form a *strong classifier* (e.g., AdaBoost)
  - More on that in AI & machine learning courses
- Training data:
  - Ca. 5000 hand labeled faces
    - Many variations (illumination, pose, skin color, …)
  - 10000 non-faces
  - Faces are normalized (scale, translation)

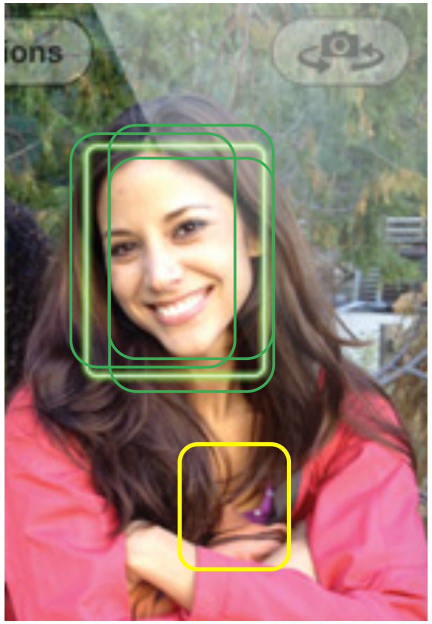
- First weak classifiers with largest weights are meaningful and have high discriminative power:
  - Eyes region is darker than the upper-cheeks
  - Nose bridge region is brighter than the eyes

- Arrange in a filter cascade:
  - Classifier with highest weight comes first
    - Or small sets of weak classifiers in one stage
  - If window fails one stage in cascade → discard window
    - Advantage: "early exit" if "clearly" non-face
  - Typical detector has 38 stages in the cascade, ~6000 features
- Effect: more features → less false positives
  - Typical visualization: Receiver operating characteristic (ROC curve)





- Final stage: only report face, if cascade finds several nearby face windows
  - Discard "lonesome" windows



Visualization of the Algorithm

Adam Harv
(http://vimeo.com/12774628)

Final remarks on Viola-Jones

- Pros:
  - Extremely fast feature computation
  - Scale and location invariant detector
    - Instead of scaling the image itself (e.g. pyramid-filters), we scale the features
  - Works also for some other types of objects
- Cons:
  - Doesn't work very well for 45° views on faces
  - Not rotation invariant