Course Evaluations

http://www.siggraph.org/courses_evaluation

4 Random Individuals will win an ATI Radeon™ HD2900XT
A Gentle Introduction to Bilateral Filtering and its Applications

- From Gaussian blur to bilateral filter – S. Paris
- Applications – F. Durand
- Link with other filtering techniques – P. Kornprobst

Implementation – S. Paris
- Variants – J. Tumblin
- Advanced applications – J. Tumblin
- Limitations and solutions – P. Kornprobst
Recap

Sylvain Paris – MIT CSAIL
Decomposition into Large-scale and Small-scale Layers

edge-preserving: Bilateral Filter
Weighted Average of Pixels

- Depends on spatial distance and intensity difference
  - Pixels across edges have almost influence

\[
BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I_p - I_q|) I_q
\]
Efficient Implementations of the Bilateral Filter

Sylvain Paris – MIT CSAIL
Outline

• Brute-force Implementation

• Separable Kernel [Pham and Van Vliet 05]

• Box Kernel [Weiss 06]

• 3D Kernel [Paris and Durand 06]
Brute-force Implementation

\[
BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(\| I_p - I_q \|) I_q
\]

For each pixel \( p \)

For each pixel \( q \)

Compute \( G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(\| I_p - I_q \|) I_q \)

8 megapixel photo: 64,000,000,000,000 iterations!

VERY SLOW!

More than 10 minutes per image
Complexity

• Complexity = “how many operations are needed, how this number varies”

• $S$ = space domain = set of pixel positions

• $|S| = \text{cardinality of } S = \text{number of pixels}$
  – In the order of 1 to 10 millions

• Brute-force implementation: $O(|S|^2)$
Better Brute-force Implementation

Idea: Far away pixels are negligible

For each pixel $p$

a. For each pixel $q$ such that $\| p - q \| < cte \times \sigma$
Discussion

• Complexity: $O(|S| \times \sigma_s^2)$

• Fast for small kernels: $\sigma_s \sim 1$ or 2 pixels

• BUT: slow for larger kernels
Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]
Separable Kernel [Pham and Van Vliet 05]

- Strategy: filter the rows then the columns

- Two “cheap” 1D filters instead of an “expensive” 2D filter
Discussion

- **Complexity:** $O(|S| \times \sigma_s)$
  - Fast for small kernels (<10 pixels)

- **Approximation:** BF kernel not separable
  - Satisfying at strong edges and uniform areas
  - Can introduce visible streaks on textured regions
separable kernel
mostly OK,
some visible artifacts
(streaks)
Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]
Box Kernel  [Weiss 06]

- Bilateral filter with a square box window  [Yarovlasky 85]

\[ Y[I]_p = \frac{1}{W_p} \sum_{q \in S} B_{\sigma_s}(\| p - q \|) G_{\sigma_r}(\| I_p - I_q \|) I_q \]

- The bilateral filter can be computed only from the list of pixels in a square neighborhood.
Box Kernel [Weiss 06]

- Idea: fast histograms of square windows

Tracking one window

**input:**
full histogram is known

**update:**
add one line, remove one line
Box Kernel [Weiss 06]
• Idea: fast histograms of square windows

Tracking two windows at the same time

input:
full histograms are known

update:
add one line, remove one line,
add two pixels, remove two pixels
Discussion

• **Complexity**: $O(|S| \times \log \sigma_s)$
  – always fast

• **Only single-channel images**

• **Exploit vector instructions of CPU**

• **Visually satisfying results (no artifacts)**
  – 3 passes to remove artifacts due to box windows (Mach bands)
brute-force implementation
box kernel visually different, yet no artifacts
Outline

• Brute-force Implementation

• Separable Kernel [Pham and Van Vliet 05]

• Box Kernel [Weiss 06]

• 3D Kernel [Paris and Durand 06]
3D Kernel [Paris and Durand 06]

- Idea: represent image data such that the weights depend only on the distance between points.

1D image

Plot

\[ I = f(x) \]
1st Step: Re-arranging Symbols

\[ BF \left[ I \right]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s} \left( \| p - q \| \right) G_{\sigma_r} \left( \| I_p - I_q \| \right) I_q \]

\[ W_p = \sum_{q \in S} G_{\sigma_s} \left( \| p - q \| \right) G_{\sigma_r} \left( \| I_p - I_q \| \right) \]

Multiply first equation by \( W_p \)

\[ W_p \cdot BF \left[ I \right]_p = \sum_{q \in S} G_{\sigma_s} \left( \| p - q \| \right) G_{\sigma_r} \left( \| I_p - I_q \| \right) I_q \]

\[ W_p = \sum_{q \in S} G_{\sigma_s} \left( \| p - q \| \right) G_{\sigma_r} \left( \| I_p - I_q \| \right) 1 \]
1st Step: Summary

\[ W_p \ BF \ [I]_p = \sum_{q \in S} G_{\sigma_s} (\| p - q \|) G_{\sigma_r} (\| I_p - I_q \|) I_q \]

\[ W_p = \sum_{q \in S} G_{\sigma_s} (\| p - q \|) G_{\sigma_r} (\| I_p - I_q \|) 1 \]

- Similar equations
- No normalization factor anymore
- Don’t forget to divide at the end
2nd Step: Higher-dimensional Space

- “Product of two Gaussians” = higher dim. Gaussian
2nd Step: Higher-dimensional Space

- 0 almost everywhere, $I$ at “plot location”
2nd Step: Higher-dimensional Space

- 0 almost everywhere, $I$ at "plot location"
- Weighted average at each point = Gaussian blur
2nd Step: Higher-dimensional Space

- 0 almost everywhere, $I$ at “plot location”
- Weighted average at each point = Gaussian blur
- Result is at “plot location”
New num. scheme:
- simple operations
- complex space

Higher dimensional functions

Gaussian blur

Higher dimensional Homogeneous intensity

division

slicing
Strategy: downsampled convolution

- Higher dimensional functions
- Downsample
- Gaussian convolution
- Upsample
- Division
- Slicing

Conceptual view, not exactly the actual algorithm
Actual Algorithm

• Never compute full resolution
  – On-the-fly downsampling
  – On-the-fly upsampling

• 3D sampling rate = \( (\sigma_s, \sigma_s, \sigma_r) \)
Pseudo-code: Start

- **Input**
  - image $I$
  - Gaussian parameters $\sigma_s$ and $\sigma_r$

- **Output:** $BF[I]$

- **Data structure:** 3D arrays $w_i$ and $w$ (init. to 0)
Pseudo-code:
On-the-fly Downsampling

- For each pixel \((X, Y) \in S\)
  
  - **Downsample:**
    \[
    (x, y, z) = \left( \left\lfloor \frac{X}{\sigma_s} \right\rfloor, \left\lfloor \frac{Y}{\sigma_s} \right\rfloor, \left\lfloor \frac{I(X,Y)}{\sigma_r} \right\rfloor \right)
    \]
    
    - **Update:**
      \[
      \begin{align*}
      w_i(x, y, z) &= I(X, Y) \\
      w(x, y, z) &= 1
      \end{align*}
      \]
Pseudo-code: Convolving

- For each axis $x$, $y$, and $z$

  - For each 3D point $(x, y, z)$

    - Apply a Gaussian mask $(1, 4, 6, 4, 1)$ to $w_i$ and $w$
      e.g., for the $x$ axis:

      \[
      w_i'(x) = w_i(x-2) + 4.w_i(x-1) + 6.w_i(x) + 4.w_i(x+1) + w_i(x+2)
      \]
Pseudo-code: On-the-fly Upsampling

- For each pixel $(X, Y) \in S$
  - Linearly interpolate the values in the 3D arrays

$$BF[I](X,Y) = \frac{\text{interpolate}(w_i, X, Y, I(X,Y))}{\text{interpolate}(w, X, Y, I(X,Y))}$$
Discussion

- **Complexity:** \( O\left( |S| + \frac{|S||R|}{\sigma^2_s \sigma_r} \right) \)

- **Fast for medium and large kernels**
  - Can be ported on GPU [Chen 07]: always very fast

- **Can be extended to color images but slower**

- **Visually similar to brute-force computation**
brute-force implementation
3D kernel
visually similar
Running Times

![Graph showing running times for different kernels: brute force, separable kernel, 3D kernel, and box kernel. The x-axis represents the spatial radius $\sigma_s$ of the kernel (in pixels) on a log scale, and the y-axis represents time (in s) on a log scale. The graph compares the performance of these kernels across different spatial radii.](image)
How to Choose an Implementation?

Depends a lot on the application. A few guidelines:

- **Brute-force**: tiny kernels or if accuracy is paramount

- **Box Kernel**: for short running times on CPU with any kernel size, e.g. editing package

- **3D kernel**:
  - if GPU available
  - if only CPU available: large kernels, color images, cross BF (e.g., good for computational photography)
Questions?