6.865 Guest Lecture
Fast Filtering
Andrew Adams
3/15/2012
Why blur an image?
Why blur an image?

• To remove noise before processing

• So we can use simpler filters later

• To decompose the input into different frequency bands
  – tonemapping, blending, etc
Computer Vision in One Slide

1) Extract some features from some images

2) Use these to formulate some (hopefully linear) constraints

3) Solve a system of equations your favorite method to produce...
Computer Vision in One Slide

0) Blur the input

1) Extract some features from some images

2) Use these to formulate some (hopefully linear) constraints

3) Solve a system of equations your favorite method to produce...
Blurry + Detail = Image
Blurry + 2 x Detail
Output = Blurry + 2 x Detail
Composing Filters

• F is a bad gradient filter
• It’s cheap to evaluate
  – $\text{val} = \text{Im}(x+5, y) - \text{Im}(x-5, y)$
Composing Filters

- F is a bad gradient filter
- It’s cheap to evaluate
  - val = Im(x+5, y) – Im(x-5, y)

- G is a good gradient filter ->
- It’s expensive to evaluate
  - for (dx=-10; dx<10; dx++)
    val += filter(dx)*Im(x+dx, y)
Composing Filters

- But $F \ast B = G$

and convolution is **associative**

- so: $G \ast \text{Im} = (F \ast B) \ast \text{Im} = F \ast (B \ast \text{Im})$
Composing Filters

• So if you need to take lots of good filters:
• Blur the image nicely once $\text{Im}_2 = (B^*\text{Im})$
• Use super simple filters for everything else

• $F_1 * \text{Im}_2 \quad F_2 * \text{Im}_2 \quad F_3 * \text{Im}_2 \quad ...$

• You only performed one expensive filter (B)
• Let’s make the expensive part as fast as possible
Fast Filtering

• Filters are used throughout image processing
• We want them to be fast
  – Even when the filter is large

\[
Out(x, y) = \sum_{u,v \in \Omega} In(x-u, y-v).filter(u, v)
\]
The Rect Filter

\[ Out(x, y) = \frac{1}{(2r+1)^2} \sum_{u,v \in [-r,r]^2} In(x-u, y-v) \]
Fast Rect Filters

10 50 50 60 10 10 10 50 40 50

\[
\begin{array}{c}
10 \\
\div 5 \\
10
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]
Fast Rect Filters

10 50 50 60 10 10 10 50 40 50

110

110 ÷ 5 = 22
Fast Rect Filters
Fast Rect Filters

\[
\begin{array}{cccccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\hline
180 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccccccccc}
22 & 28 & 36 & \\
\hline
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\hline
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]

\[
\begin{array}{c}
- \\
\end{array}
\]

\[
\begin{array}{c}
+ \\
\end{array}
\]

\[
\begin{array}{c}
\div 5 \\
\end{array}
\]

\[
\begin{array}{cccc}
22 & 28 & 36 & 36 \\
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]

\[
\begin{array}{cccc}
22 & 28 & 36 & 36 & 28 & 28 & \\
\end{array}
\]

\[
\begin{array}{c}
\div 5
\end{array}
\]

\[
\begin{array}{c}
140
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\downarrow & - & & & & & & & + & \\
\downarrow & & & & & & & & 120 & \\
\downarrow & \div 5 & & & & & & & & \\
22 & 28 & 36 & 36 & 28 & 28 & 24 & & & \\
\end{array}
\]
Fast Rect Filters

\[
\begin{array}{cccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
22 & 28 & 36 & 36 & 28 & 28 & 24 & 32 & \\
\end{array}
\]

\[\div 5\]

\[\div 5\]

\[160\]
Fast Rect Filters

\[
\begin{array}{cccccccccc}
10 & 50 & 50 & 60 & 10 & 10 & 10 & 50 & 40 & 50 \\
\end{array}
\]

\[
\frac{150}{5}
\]

\[
\begin{array}{cccccccccc}
22 & 28 & 36 & 36 & 28 & 28 & 24 & 32 & 30 \\
\end{array}
\]
Fast Rect Filters

10  50  50  60  10  10  10  50  40  50

-  

140  ÷5

22  28  36  36  28  28  24  32  30  28
Fast Rect Filters

• Complexity?
  – Horizontal pass: $O((w+f)h) = O(wh)$
  – Vertical pass: $O((h+f)w) = O(wh)$
  – Total: $O(wh)$

• Precision can be an issue

\[
\begin{array}{cccccccc}
0 & 0 & 5^{10} & 5 & 0 & 0 & 0 & 0 \\
5^9 & 5^9 & 5^9 & 5^9 & 5^9 & 0 & -1 & -1 \\
\end{array}
\]
Fast Rect Filters

• How can I do this in-place?
Gaussian Filters

\[ \text{Out}(x, y) = \left( \sum_{u,v \in [-r,r]^2} \text{In}(x-u, y-v) \cdot e^{(u^2+v^2)/2\sigma^2} \right) \]
Gaussian Filters

• How can we extend this approach to Gaussian filters?
• Common approach:
  – Take FFT $O(wh \ln(w) \ln(h))$
  – Multiply by FFT of Gaussian $O(wh)$
  – Take inverse FFT $O(wh \ln(w) \ln(h))$
  – Total cost: $O(w \ln(w) h \ln(h))$
• Cost independent of filter size 😊
• Not particularly cache coherent 😞
Gaussian v Rect
Gaussian v Rect*Rect
Gaussian v Rect\(^3\)
Gaussian v Rect$^4$
Gaussian v Rect$^5$
Gaussian
Rect (RMS = 0.00983)
Gaussian
Rect² (RMS = 0.00244)
Gaussian
Gaussian
Rect^4 (RMS = 0.00176)
Gaussian
Rect^5 (RMS = 0.00140)
Gaussian Filters

• Conclusion: Just do 3 rect filters instead
• Why does this work?
• Cost: $O(wh)$
• Cost independent of filter size 😊
• More cache coherent 😊
• Be careful of edge conditions 😞
• Hard to construct the right filter sizes: 😞
Filter sizes

• Think of convolution as randomly scattering your data around nearby
• Convolving = summing random variables
• How far data is scattered is described by the standard deviation of the filter/distribution
  • standard deviation = sqrt(variance)
• When summing random variables, variance adds
  – Convolving by a filter with variance v twice produces a filter with variance 2v
Filter sizes

• Variance adds
  – Performing a filter with variance $v$ twice produces a filter with variance $2v$

• Standard deviation scales
  – A filter with standard deviation $s$, when scaled to be twice as wide, has standard deviation $2s$
## A Gaussian from three Rects

<table>
<thead>
<tr>
<th>Width of rect</th>
<th>Variance of three rects</th>
<th>Std.dev. of equivalent Gaussian</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1.41</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2.45</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>3.46</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>4.47</td>
</tr>
<tr>
<td>11</td>
<td>30</td>
<td>5.47</td>
</tr>
<tr>
<td>13</td>
<td>42</td>
<td>6.48</td>
</tr>
<tr>
<td>15</td>
<td>56</td>
<td>7.48</td>
</tr>
<tr>
<td>17</td>
<td>72</td>
<td>8.48</td>
</tr>
</tbody>
</table>
Integral Images

- Fast rects are good for filtering an image...
- But what if we need to compute lots of filters of different shapes and sizes quickly?

- Classifiers need to do this
Integrate the Image ahead of time

\[ \text{Integral}(x, y) = \sum_{u,v=(0,0)}^{(x,y)} \text{Input}(u, v) \]

- Each pixel is the sum of everything above and left
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters
Integral Images

• Fast to compute (just run along each row and column adding up)

• Allows for arbitrary sized rect filters
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters
Integral Images

• Fast to compute (just run along each row and column adding up)
• Allows for arbitrary sized rect filters
Integral Images

• Fast sampling of arbitrary rects
• Precision can be an issue
• Can only be used for rect filters...
Higher Order Integral Images

• Fast sampling of arbitrary polynomials

\[
\text{Integral}_0(x) = \sum_{u=0}^{x} \text{Input}(u)
\]

\[
\text{Integral}_1(x) = \sum_{u=0}^{x} \text{Input}(u) \cdot u
\]

\[
\text{Integral}_2(x) = \sum_{u=0}^{x} \text{Input}(u) \cdot u^2
\]
Higher Order Integral Images

- Let’s say we want to evaluate a filter shaped like \((4-x^2)\) centered around each pixel.
Higher Order Integral Images

\[ Out(x) = \sum_{u=x-2}^{x+2} I(u)(4 - (u - x)^2) \]

\[ O(x) = (4 - x^2) \sum_{u=x-2}^{x+2} I(u) + 2x \sum_{u=x-2}^{x+2} uI(u) + \sum_{u=x-2}^{x+2} u^2 I(u) \]
Higher Order Integral Images

We can compute each term using the integral images of various orders.

No summations over $u$ required.

$$O(x) = (4 - x^2) \sum_{u=x-2}^{x+2} I(u)$$

$$+ 2x \sum_{u=x-2}^{x+2} uI(u)$$

$$+ \sum_{u=x-2}^{x+2} u^2 I(u)$$
Gaussians using Higher Order Integral Images

• Construct a polynomial that looks kinda like a Gaussian, e.g. \((x-1)^2(x+1)^2\)
IIR Filters

• We can also use feedback loops to create Gaussians...
IIR Filters

0 0 64 0 0 0 0 0 0 0 0

÷2

0
IIR Filters

\[ \begin{array}{cccccccccc}
0 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]
IIR Filters

\[
\begin{array}{cccccccccc}
0 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
+ & + & \div 2 & +
\end{array}
\]

\[
\begin{array}{ccccccc}
0 & 0 & 32 & & & & \\
\end{array}
\]
IIR Filters

\[
\begin{array}{cccccccccccc}
0 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
0 & 0 & 32 & 16 & & & & & & & & \\
\end{array}
\]
IIR Filters
IIR Filters

\[
\begin{array}{cccccccccc}
0 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccccccccc}
0 & 0 & 32 & 16 & 8 & 4 & \text{⋯} & \text{⋯} & \text{⋯} & \text{⋯} & \text{⋯}
\end{array}
\]

\[
\begin{array}{c}
+ \\
+ \\
\div 2
\end{array}
\]
IIR Filters

0 0 64 0 0 0 0 0 0 0 0

0 0 32 16 8 4 2
IIR Filters

\[
\begin{array}{ccccccccccc}
0 & 0 & 64 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 32 & 16 & 8 & 4 & 2 & 1 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\div 2 & + & + & + & \\
\end{array}
\]
IIR Filters

0 0 64 0 0 0 0 0 0 0 0

0 0 32 16 8 4 2 1 0.5 0.25
IIR = Infinite Impulse Response

- A single spike has an effect that continues forever
- The example above was an exponential decay
- Equivalent to convolution by:
IIR Filters

- Can be done in place 😊
- Makes large, smooth filters, with very little computation 😊
- Somewhat lopsided...
IIR Filters

- One forward pass, one backward pass
- = exponential decay convolved with flipped exponential decay
- Somewhat more Gaussian-ish
More Advanced IIR Filters

Each output is a weighted average of the next input, and the last few outputs.
More Advanced IIR Filters

• It’s possible to optimize the parameters to match a Gaussian of a certain std.dev.
• It’s harder to construct a family of them that scales across standard deviations
  – (see ImageStack’s filter.cpp)
How can we convolve by this filter?
Filtering by Resampling

• This looks like we just zoomed a small image

• Can we filter by downsampling then upsampling?
Filtering by Resampling
Filtering by Resampling

- Downsampled with rect (averaging down)
- Upsampled with linear interpolation
Use better upsampling?

• Downsampling with rect (averaging down)
• Upsampled with bicubic interpolation
Use better downsampling?

• Downsamped with tent filter
• Upsampled with linear interpolation
Use better downsampling?

• Downsampling with bicubic filter
• Upsampling with linear interpolation
Resampling Simulation

- http://people.csail.mit.edu/abadams/resampling.swf
Best Resampling

• Downsampling, blurred, then upsampled with bicubic filter
Best Resampling

• Equivalent to downscaled, then upscaled with a blurred bicubic filter
What's the point?

• If we can blur quickly without resampling, why bother resampling?
What's the point?

• If we can blur quickly without resampling, why bother resampling?

• A: Memory use

• Store the blurred image at low res, sample it at higher res as needed.
Recap: Fast Linear Filters

1) Separate into a sequence of simpler filters
   - e.g. Gaussian is separable across dimension
   - and can be decomposed into rect filters

2) Separate into a sum of simpler filters
Recap: Fast Linear Filters

3) Separate into a sum of easy-to-precompute components (integral images)
   - great if you need to compute lots of different filters

4) Resample
   - great if you need to save memory

5) Use feedback loops (IIR filters)
   - great, but hard to change the std.dev of your filter
Histogram Filtering

• The fast rect filter
  – maintained a sum
  – updated it for each new pixel
  – didn't recompute from scratch

• What other data structures might we maintain and update for more complex filters?
Histogram Filtering

• The min filter, max filter, and median filter
  – Only care about what pixel values fall into neighbourhood, not their location
  – Maintain a histogram of the pixels under the filter window, update it as pixels enter and leave
Histogram Updating
Histogram Updating
Histogram-Based Fast Median

• Maintain:
  – hist = Local histogram
  – med = Current Median
  – lt = Number of pixels less than current median
  – gt = Number of pixels greater than current median
Histogram-Based Fast Median

• while (lt < gt):
  – med--
  – Update lt and gt using hist

• while (gt < lt):
  – med++
  – Updated lt and gt using hist
Histogram-Based Fast Median

- Complexity?
- Extend this to percentile filters?
- Max filters? Min filters?
Use of a min filter: dehazing
Large-support min filter
Difference (brightened)
Weighted Blurs

- Perform a Gaussian Blur weighted by some mask
- Pixels with low weight do not contribute to their neighbors
- Pixels with high weight do contribute to their neighbors
Weighted Blurs

• Can be expressed as:

\[
O(x) = \sum_{x'=x-f}^{x+f} I(x') \cdot e^{-(\sigma_1 (x-x')^2)} \cdot w(x')
\]

• Where \( w \) is some weight term

• How can we implement this quickly?
Weighted Blurs

• Use homogeneous coordinates for color!
  – On Tuesday we used them for geometry

• Homogeneous coordinates uses (d+1) values to represent d-dimensional space

• All values of the form \([a.r, a.g, a.b, a]\) are equivalent, regardless of \(a\).

• To convert back to regular coordinates, divide through by the last coordinate
Weighted Blurs

• This is red: [1, 0, 0, 1]
• This is the same red: [37.3, 0, 0, 37.3]
• This is dark cyan: [0, 3, 3, 6]
• This is undefined: [0, 0, 0, 0]
• This is infinite: [1, 5, 2, 0]
Weighted Blurs

- Addition of homogeneous coordinates is *weighted averaging*

- \[
  \begin{bmatrix}
    x.r_0 & x.g_0 & x.b_0 & x \\
    y.r_1 & y.g_1 & y.b_1 & y
  \end{bmatrix}
  + \begin{bmatrix}
    y.r_1 & y.g_1 & y.b_1 & y
  \end{bmatrix}
  = \frac{x.r_0+y.r_1}{x+y} & \frac{x.g_0+y.g_1}{x+y} & \frac{x.b_0+y.b_1}{x+y} & x+y
\]
Weighted Blurs

• Often the weight is called alpha and used to encode transparency, in which case this is known as “premultiplied alpha”.

• We’ll use it to perform weighted blurs.
Weight:
Result:
Result:

- Why bother with uniform weights?
- Well... at least it gets rid of the sum of the weights term in the denominator of all of these equations:

\[ O(x) = \sum_{x' = x-f}^{x+f} I(x') e^{-\left(\sigma_1 (x-x')^2\right)} \]
Weight:
Result: Like a max filter but faster
Weight:
Result: Like a min filter but faster
Result: A blur that ignores the dog
Seamless Cloning
Seamless Cloning
Copy-paste doesn’t work
Separate into base + detail
Separate into base + detail
Seamless Cloning
Key Idea

• Filtering is $O(w*h)$
• No dependence on filter size