

ACM SIGGRAPH@UIUC

Fast Image Convolutions

by: Wojciech Jarosz

Image Convolution

- Traditionally, image convolution is performed by what is called the sliding window approach.
 - For each pixel in the image, a local neighborhood of pixels is multiplied by a weighting kernel, and then added up to get the value of the new pixel.
- In the most general case, the convolution of a 2D image is $O(m^2 * n^2)$, where m is the width and height of the image, and n is the width and height of the convolution kernel.
 - This is quadratic in terms of image dimension and quadratic in terms of kernel dimension!

Naïve Implementation

```
for (each pixel location p, in the old image)
{
    accumulation = 0
    weightsum = 0
    for (each pixel k, in the neighborhood of p)
    {
        accumulation += k*weight(k)
        weightsum += weight(k)
    }

    p, in new image = accumulation/weightsum
}
```

The Weighting Kernel

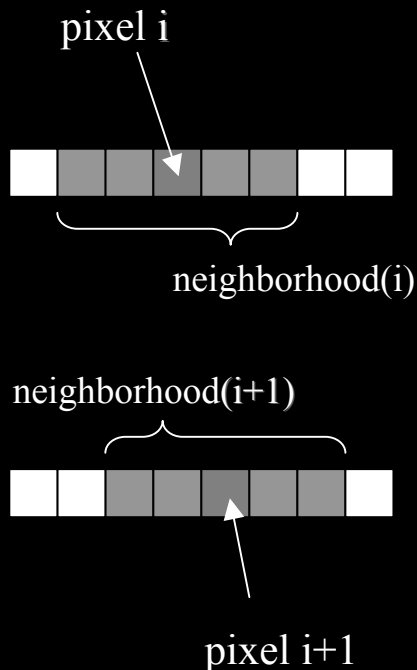
- The weighting kernel is what will determine the properties of the convolution.
- For a simple box blur, all the weights are 1.
- For a Gaussian blur, the weights fall off according to a normal distribution away from the center of the kernel.
- A simple sharpen filter would have negative weights at nearby pixels, but a positive weight at the center.

Speeding Up a Box Blur

- Instead doing a 2D box blur on an image, you can first do a horizontal motion blur, and then a vertical motion blur.
- This actually create an image that is equivalent to a box blurred image! Try it!
- This speeds it up from $O(m^2*n^2)$ to $O(m^2*2n)$!
- Quite a good speed-up, but we can still do better!

Speeding Up a Box Blur

Part 2



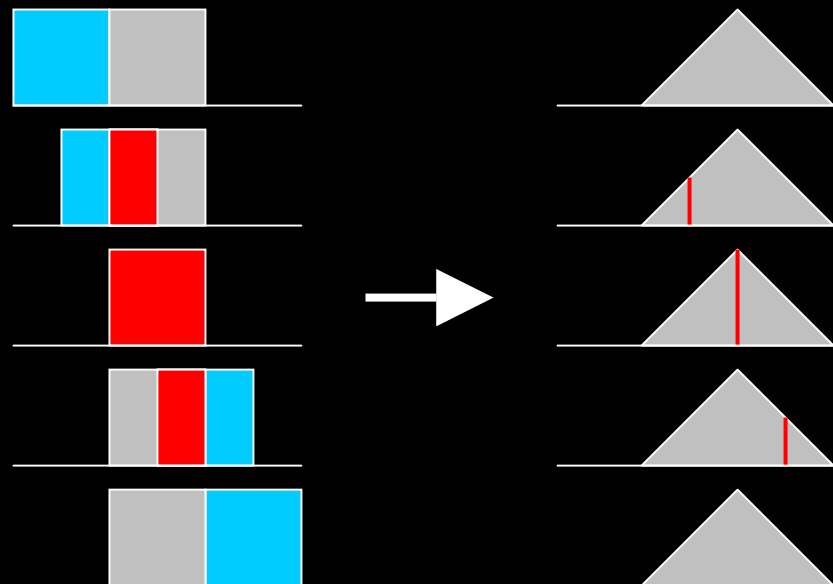
- Since we are now just doing 2 motion blurs, let's just consider motion blur:
- The accumulation of the neighborhood of pixel i , shares a lot of pixels in common with the accumulation for pixel $i+1$.
- In fact:
 - `accumulation(i+1) = accumulation(i) - leftmost pixel of neighborhood(i) + leftmost pixel of neighborhood(i+1)`
- This means we need to compute the whole kernel for only the first pixel in each row. Successive pixel blur values can be attained with just an add and a subtract to the previous blur value!
- Now it's $O(m^2)$, only dependent on image resolution! Independent of blur size!

What's next?

- We now have a box blur that is independent of blur width. What else could we ask for?
- Well, the box filter is not a very good blur kernel, we would like to use some better convolution kernels.
- How can we apply these concepts to a Gaussian blur kernel for instance?
- In order to answer this, lets first review some of the math behind convolutions ...

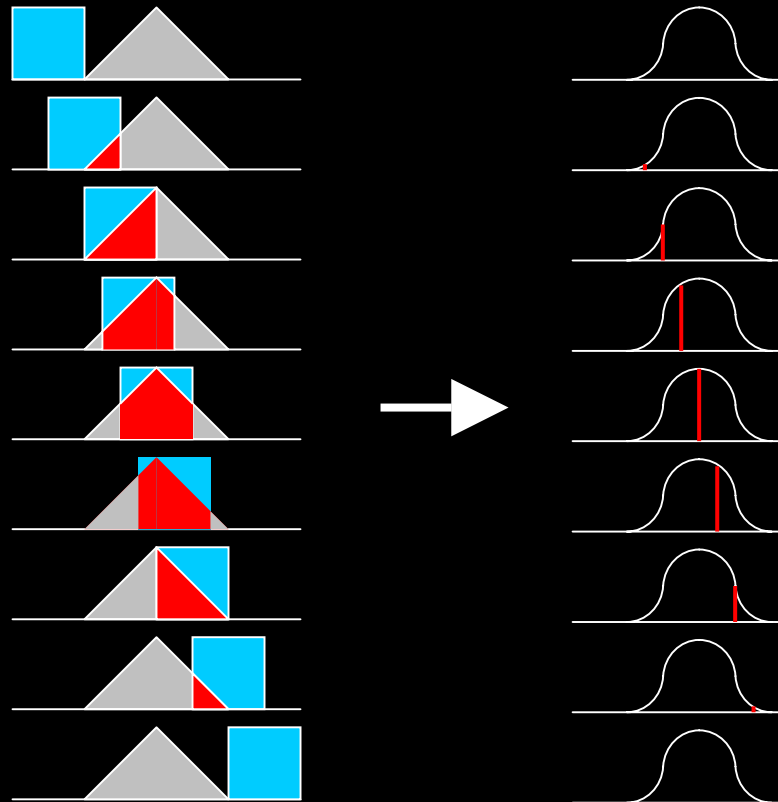
The 1D case

- Lets look at convolution in 1D for simplicity.
- The sliding window is an intuitive way to visualize convolution.
- Convolution of two square waves (box filters) yields a triangle wave (tent filter, piecewise linear).



The 1D case

- Convolution of a box filter (piecewise constant) with a tent filter (piecewise linear) yields a piecewise quadratic filter.



Taking it Further

- The pattern continues. Box filtering the piecewise quadratic curve from the last slide will yield a piecewise cubic (Bernstein polynomials, NURBS).
- Each time we make the curve more “smooth.”
- Taking this to the limit will produce a Gaussian distribution.

The Implications

- How does this relate to our 2D image blurs?
- We can put our fast box blur function to use in order to approximate a Gaussian blur!
- Applying our box filter two times will produce a tent filter, three times a piecewise quadratic, four times...
- We can therefore create a good approximation to a Gaussian blur that is still independent of radius! Its only dependent on the image size and the number of iterations we apply the box blur.

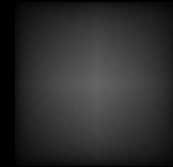
2D Filters



Motion Blur



Box Blur



Tent Blur
(box 2 twice)



Piecewise Quadratic
(box blur 3 times)



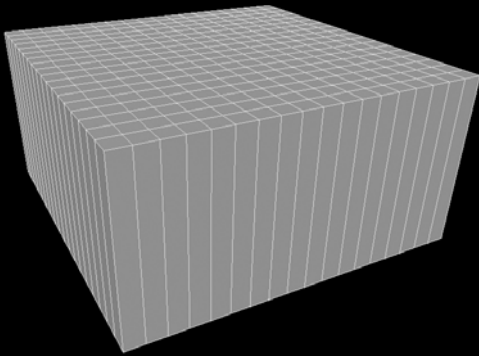
Piecewise hexic?
(box blur 6 times)



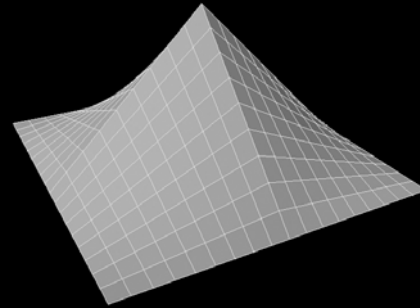
Gaussian Blur



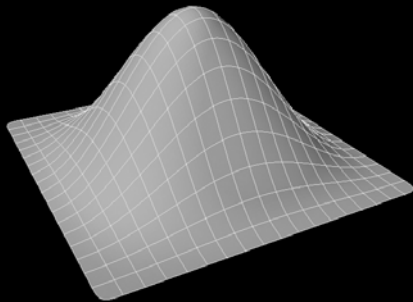
3D Visualization of 2D Filters



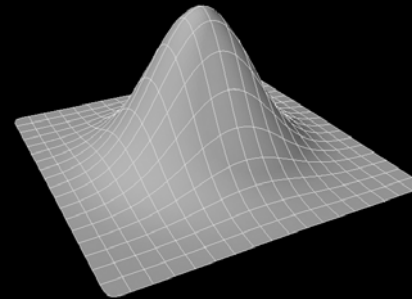
Piecewise Constant (Box) Filter



Piecewise Linear (Tent) Filter



Piecewise Quadratic
Filter



Gaussian Filter

Another Way



- Although it is still constant in time relative to the radius, to create a very smooth Gaussian approximation, many iterations are needed with this approach. In situations that a very nice blur is required, another method might be preferable.
- Our second speedup doesn't seem possible with anything but a box filter. The box filter was unique in that all its weights were equal, and that allowed us to just add a value and subtract a value to the accumulation for each pixel location.
- However, it turns out that our first speedup, doing two motion blurs, will work with other kernels as well!

Other Kernels

- A problem arises, however, with the axis aligned nature of doing just two motion blurs. Using a tent filter for each motion blur will not create a nice radial tent filter (cone filter), but a normal, axis aligned tent. The same goes for any other kernel, it will have distinct axis aligned artifacts...
- With the exception of a Gaussian! The Gaussian has the unique quality of being the same whether it is defined along the radius, or along the X and Y axes separate.
- This means that if we do two Gaussian weighted motion blurs, this will create a *radially* symmetrical Gaussian kernel!

Wrap Up

- Following these simple tips you can create image convolution routines that are orders of magnitude faster than the naïve implementations.
- Another thing to keep in mind is to pre-compute expensive kernel.
 - If you create a fast Gaussian blur function using two successive motion blurs, but you evaluate the Gaussian function every time you need to figure out a weight, you will see very little speedup.
 - Pre-compute the kernel! With our method, a radius 10 blur, would only require pre-computing/storing 11 values for the weights, since we are doing it in 1D each time.