GDC

Circular Separable Convolution Depth of Field "Circular Dof"

Kleber Garcia
Rendering Engineer – Frostbite Engine
Electronic Arts Inc.

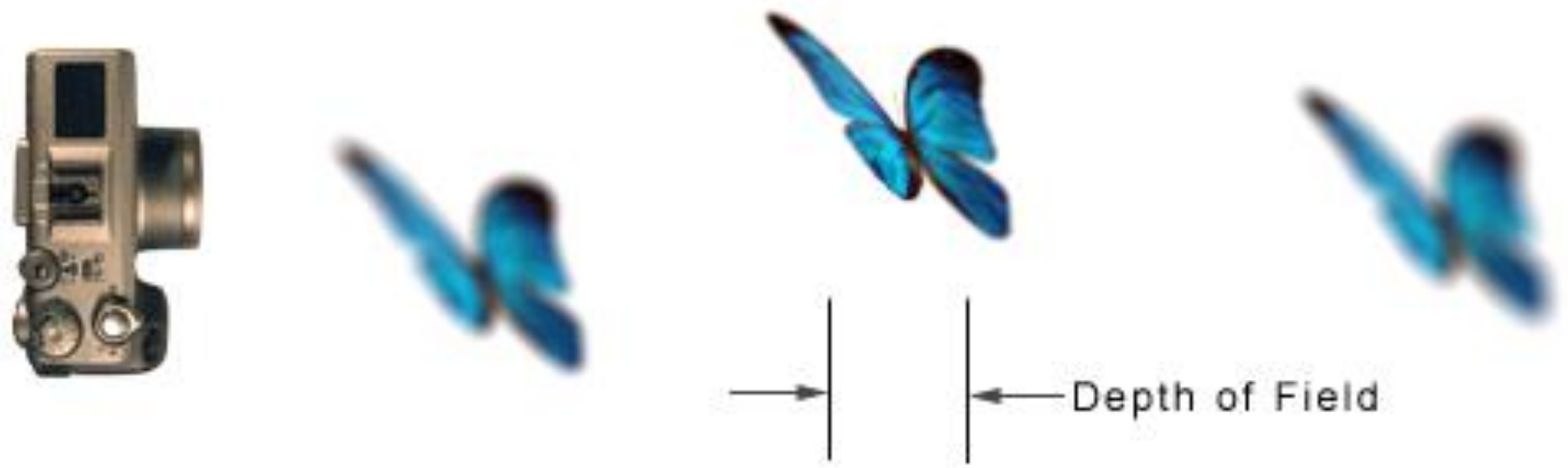
UBM

Agenda

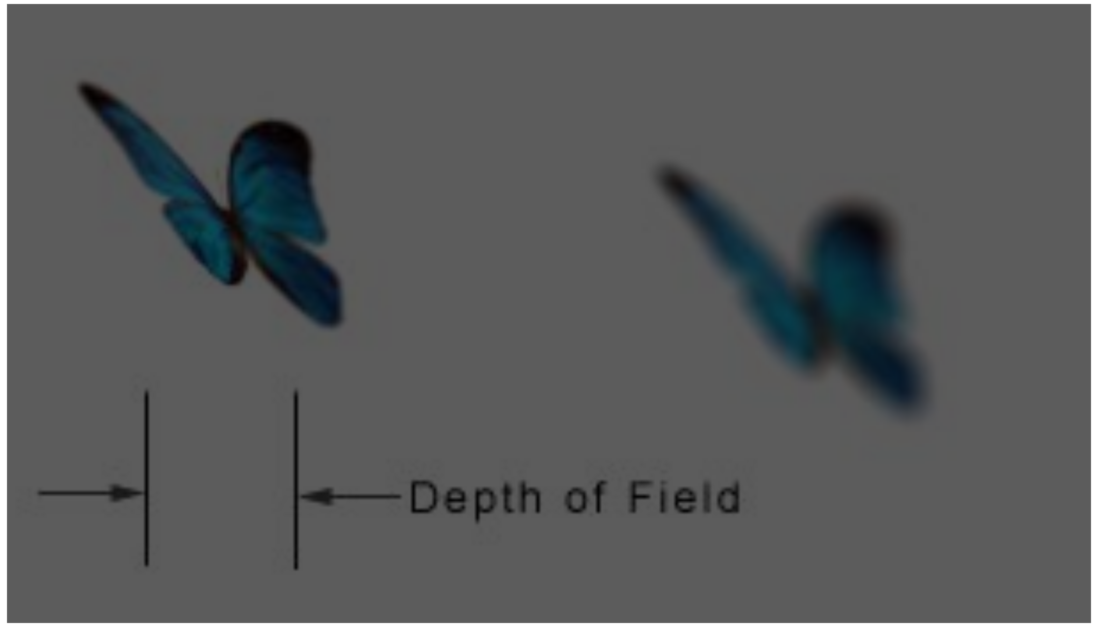
- Background
- Results
- Algorithm
- Performance analysis
- Artifacts/Improvments
- Sources / Credits
- Q/A

FROSTBITE™

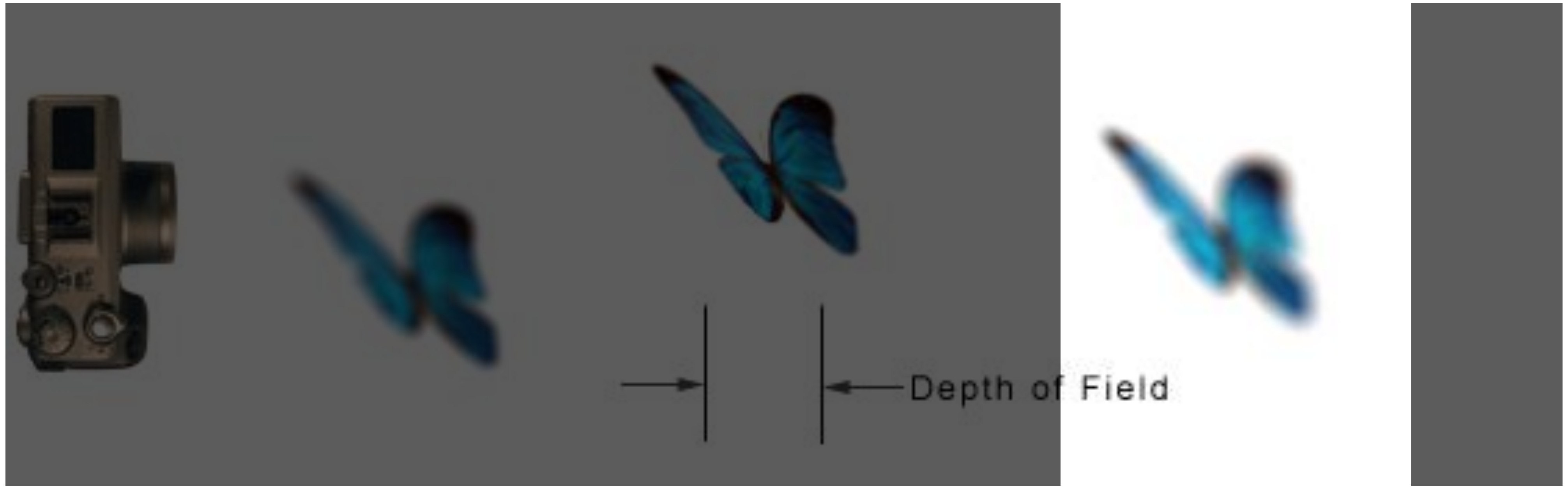
Background



Background



Background



Background


Bokeh –is the aesthetic quality of the blur produced in the out-of-focus parts of an image produced by a lens.

Background

- Circle of Confusion (COC) – optical spot caused by a cone of light rays from a lens not coming to a perfect focus when imaging a point source.
- Can be thought of the ‘radius of the blur’ at a given pixel.

Visual Target


MOSES
86
57
59
EA SPORTS
MADDEN
NFL


EA SPORTS
PGA TOUR


EA SPORTS
PGA TOUR


FROSTBITE
EA SPORTS
FIFA 17


Sprite DOF


Circular DOF

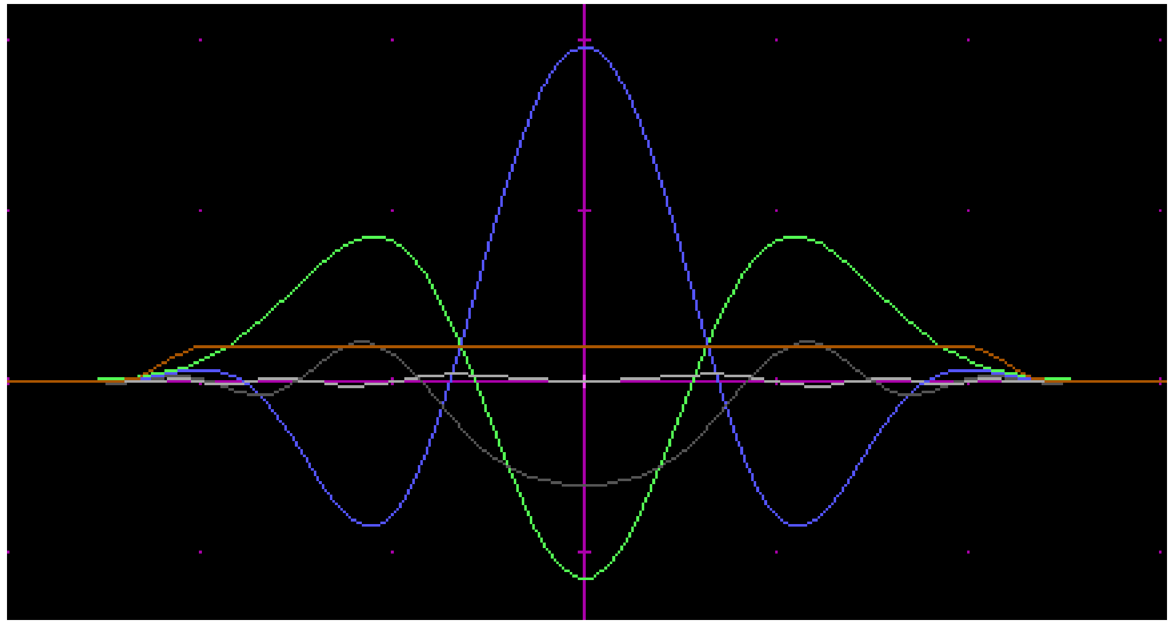
Algorithm

Separable gather (Circular Dof filter)

- http://yehar.com/blog/?p=1495

By Olli Niemitalo

- Separable circular filter.
- Possible in the frequency domain (imaginary space)!
- Decompose frame buffer into a Fourier Transform
  - Possible to mix the signals and get a circular convolution

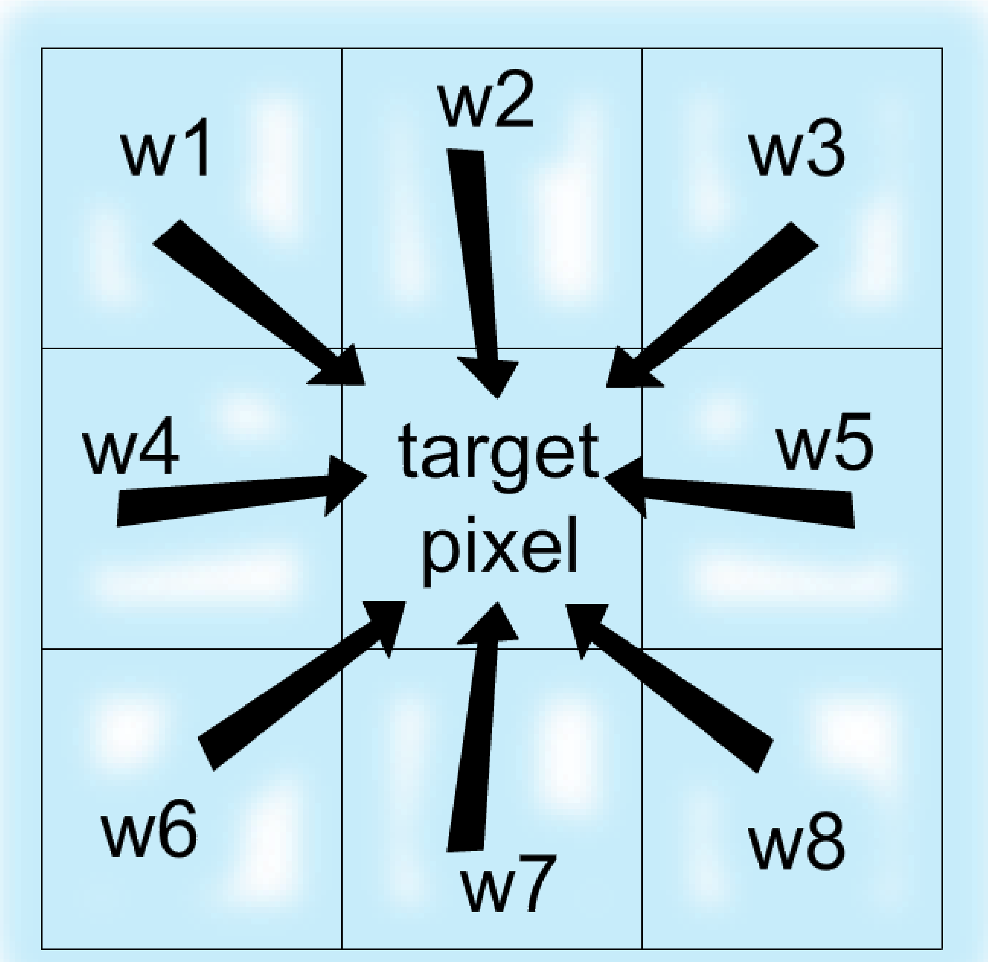


Algorithm

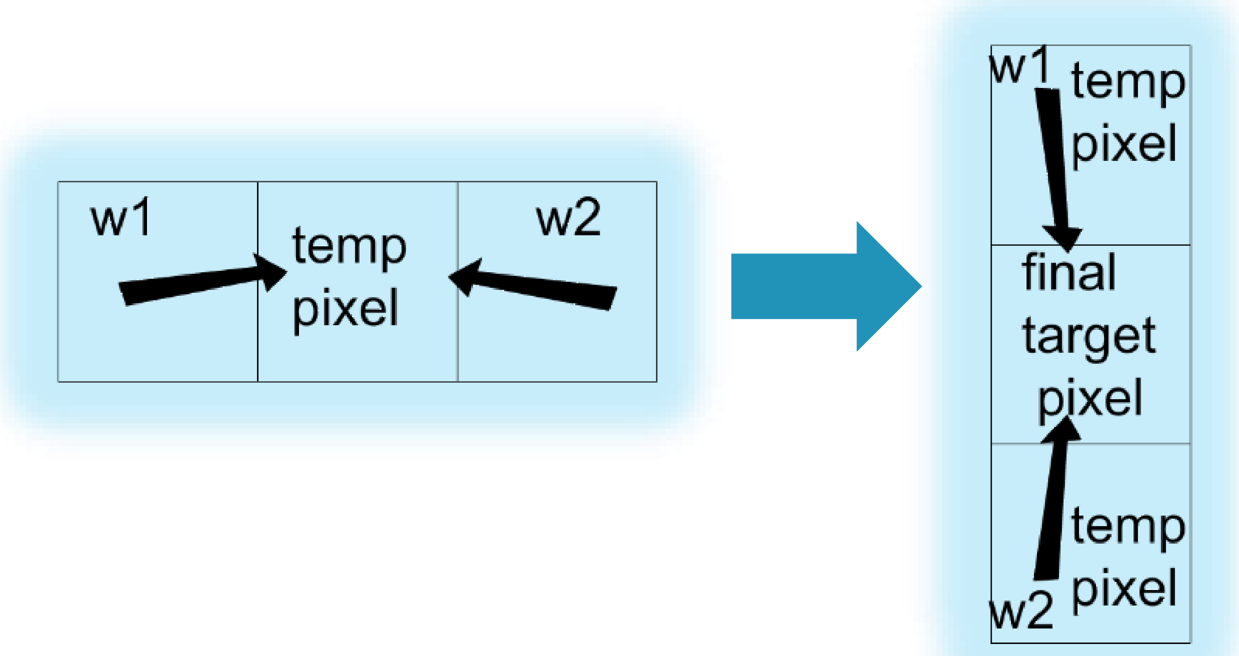
- **To understand the separable property of Circular Dof, lets first take a look at how separable Gather works.**

Algorithm

Brute force gather vs Separable Gather



Brute Force Gather - $O(n^2)$



Separable Gather - O(n)

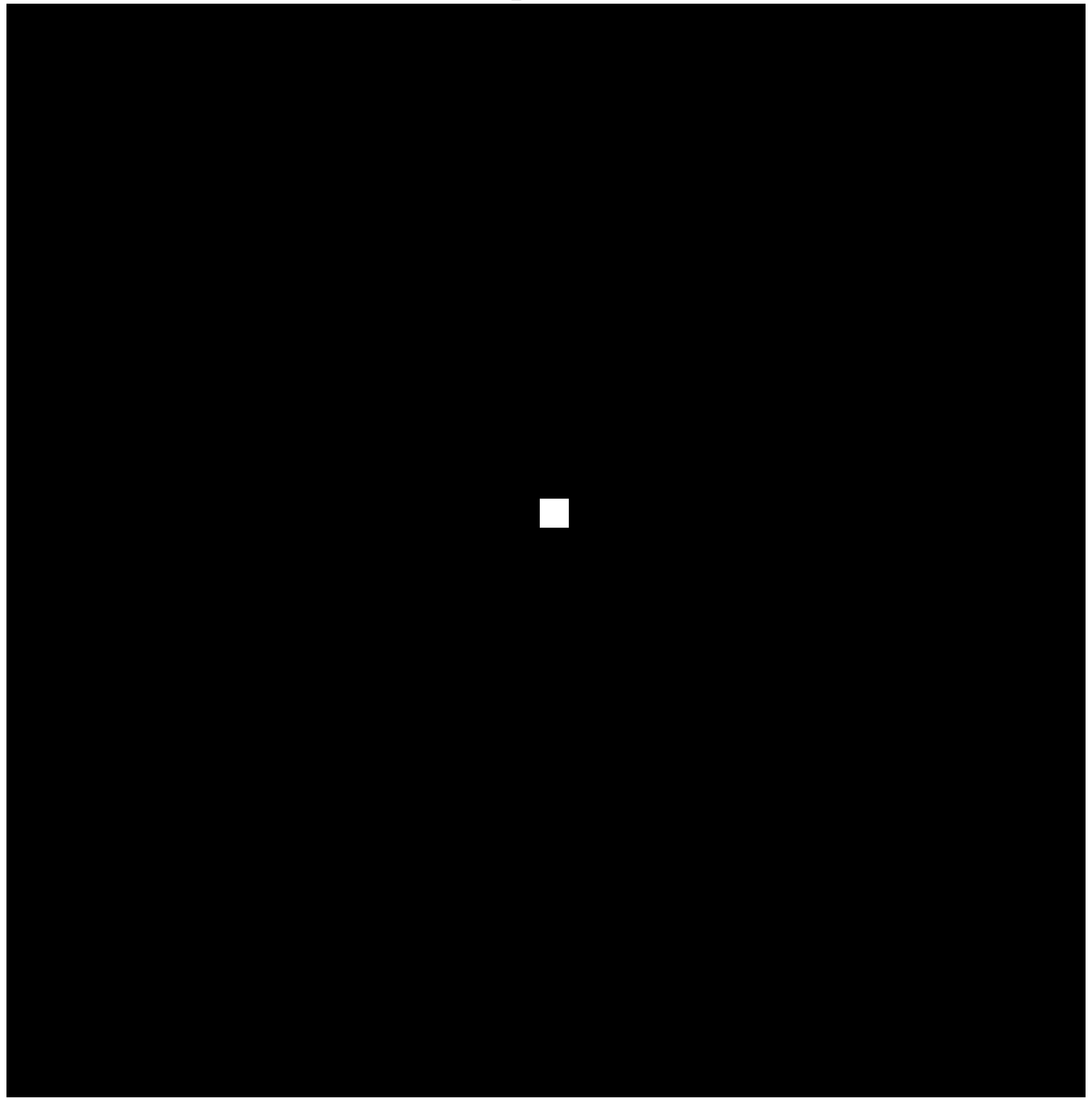
Algorithm

Separable Bokeh

- Our approach has same time complexity as separable Gather-Gaussian.

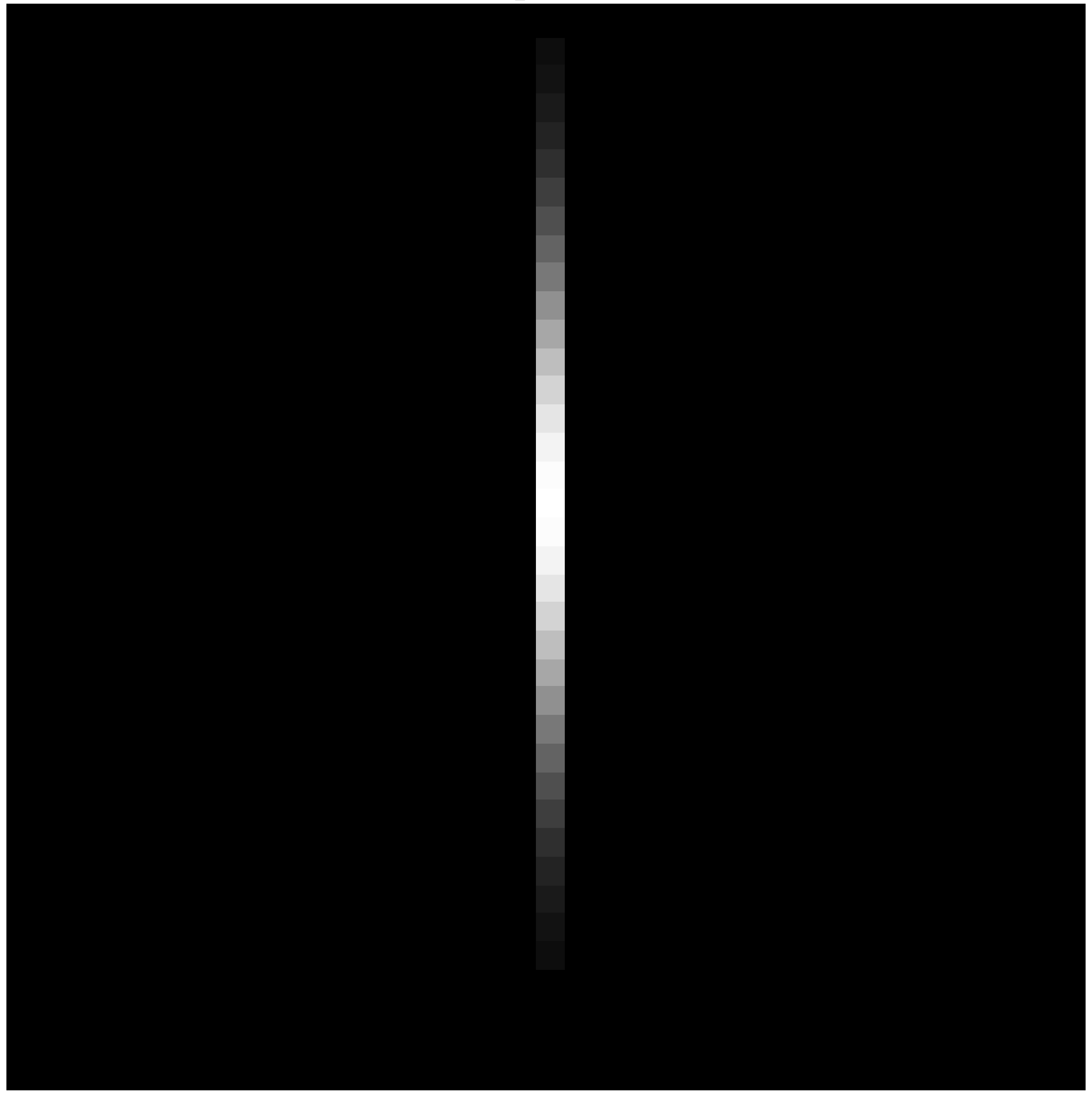
Algorithm

Separable gather (Gaussian filter)



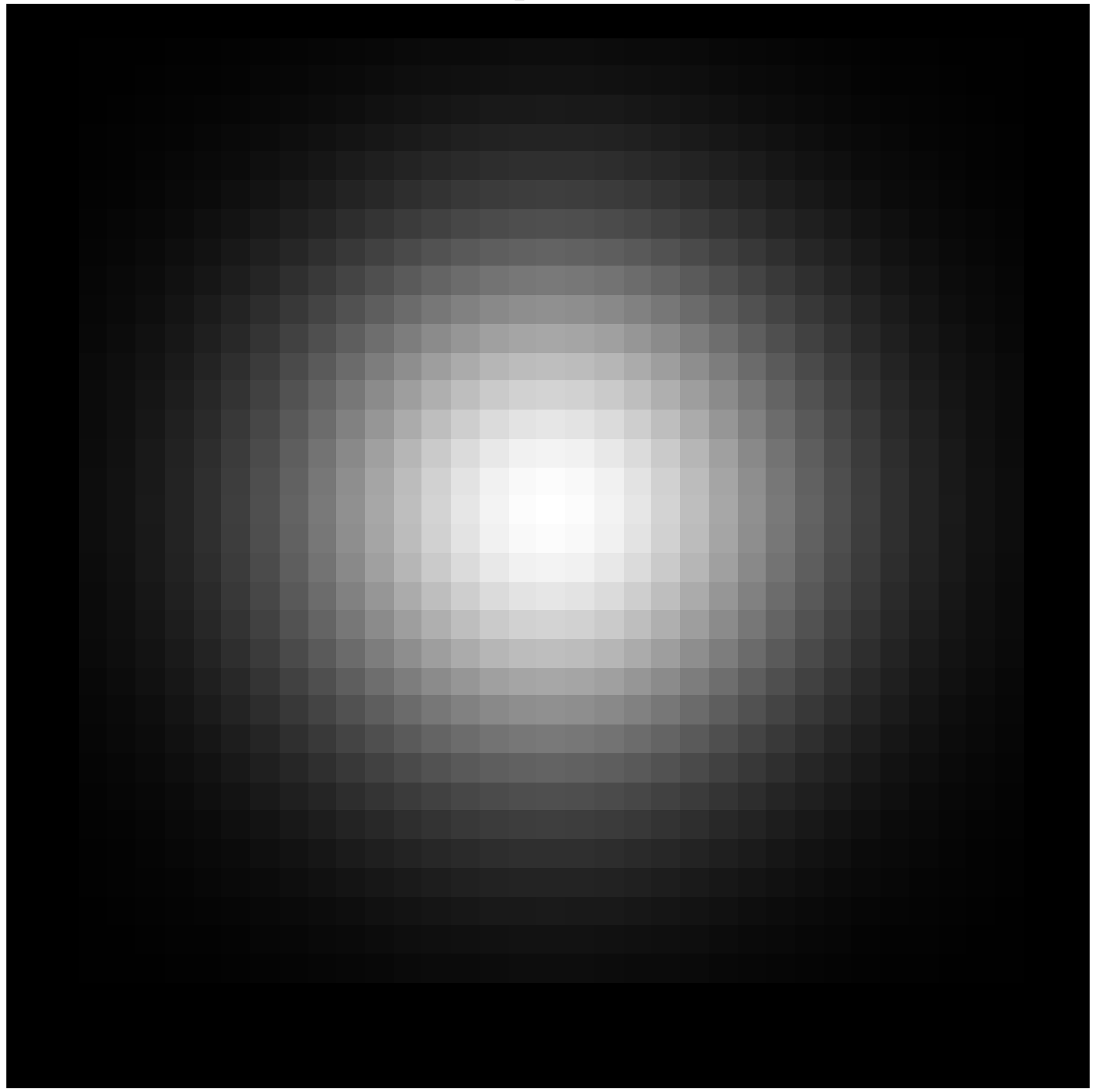
Algorithm

Separable gather (Gaussian filter)



Algorithm

Separable gather (Gaussian filter)



Algorithm

Separable gather (Gaussian filter)

- $F(x) = e^{-ax^2}$

Clear Image

Vertical Blur

Horizontal Blur

Algorithm

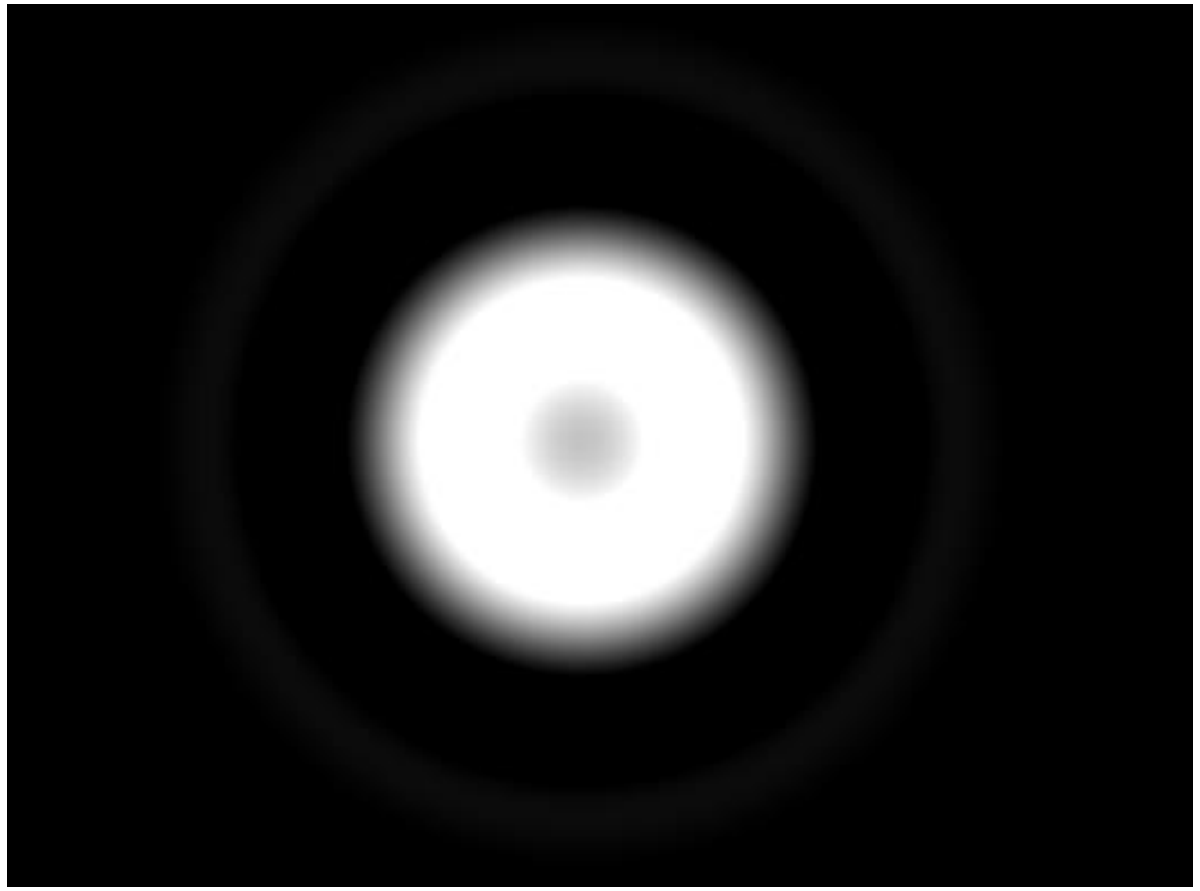
- A filter F(x) can be resolved to a set of weights.
- Our approach resolves a complex filter into a complex number
- Complex numbers have 2 components, real and imaginary
- Remember i * i = -1
- Let P be a complex number, $P = P_r + P_i i$
- The sum of two complex numbers P and Q would be
$$P + Q = (P_r + Q_r) + (P_i + Q_i)i$$
- The product of two complex numbers would be
$$P * Q = (P_r * Q_r) - (P_i * Q_i) + [(P_r * Q_i) + (Q_r * P_i)]i$$

Algorithm

- **Lets look now at circular DoF in action…**

Algorithm

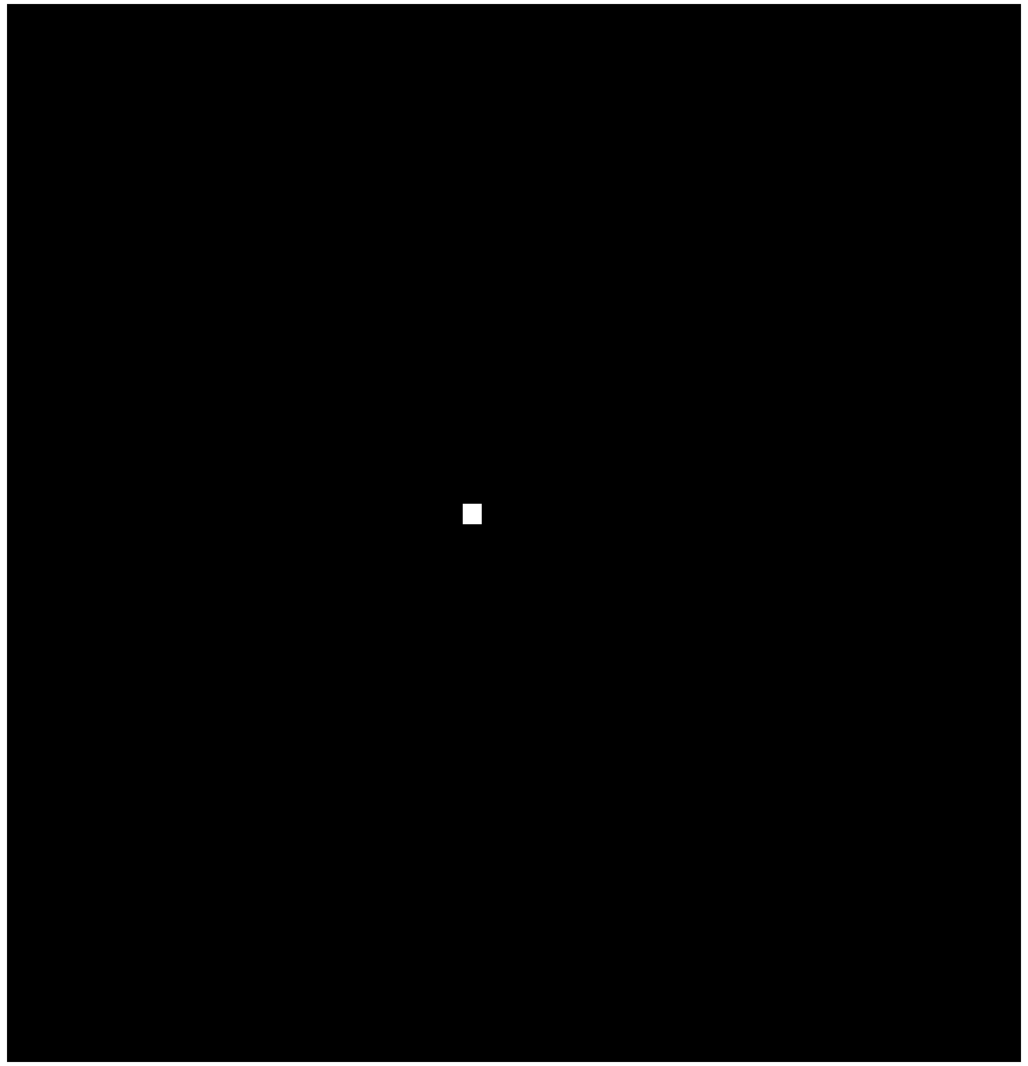
Separable gather (Circular Dof filter)



One Component Filter

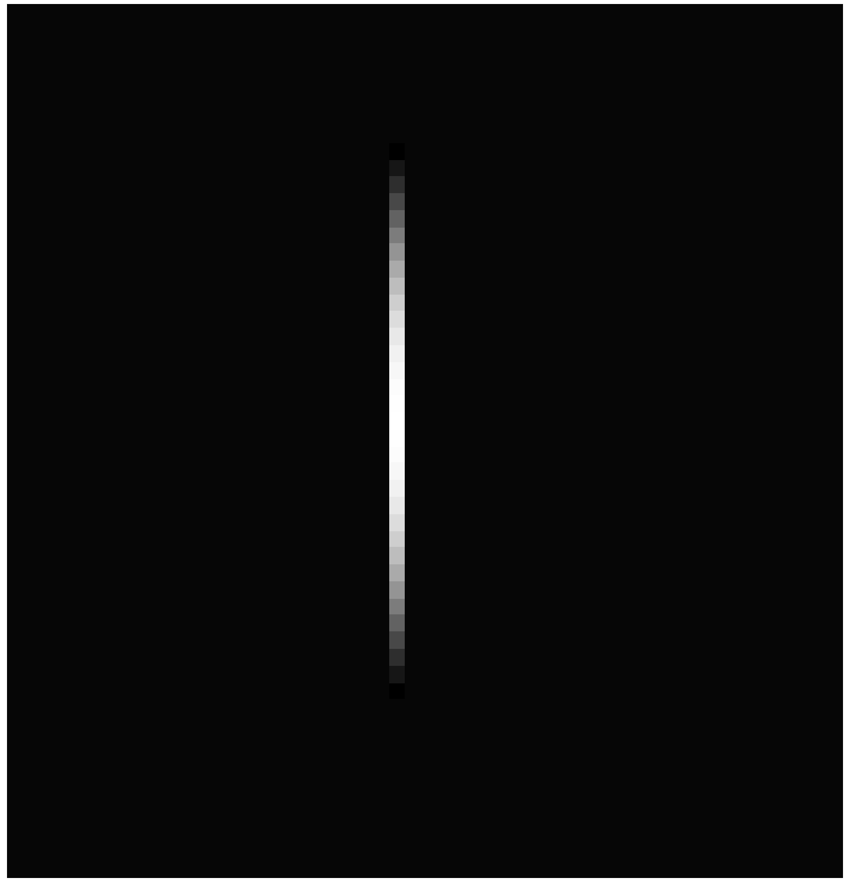
Algorithm

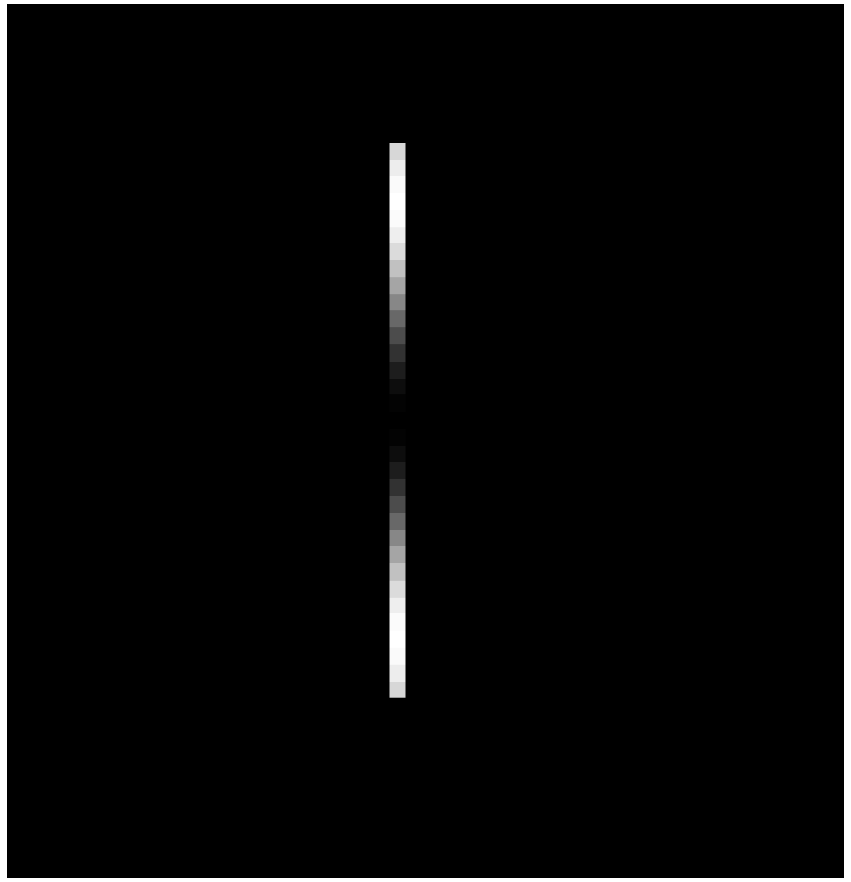
Separable gather (Circular Dof filter)



Algorithm

Separable gather (Circular Dof filter)

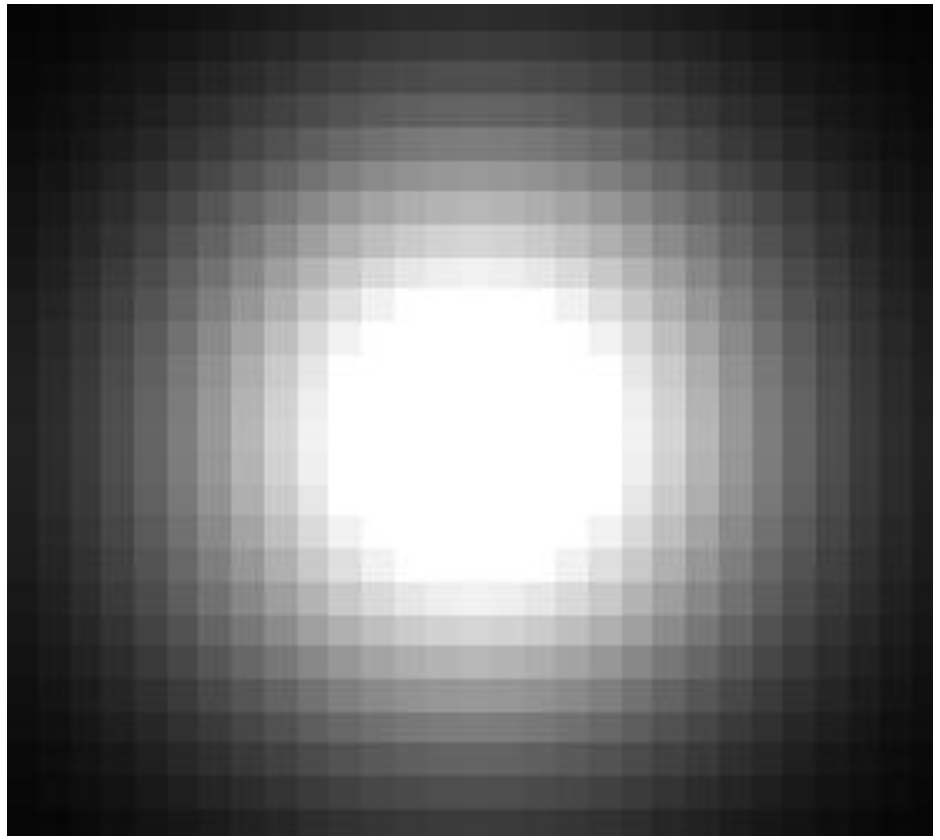


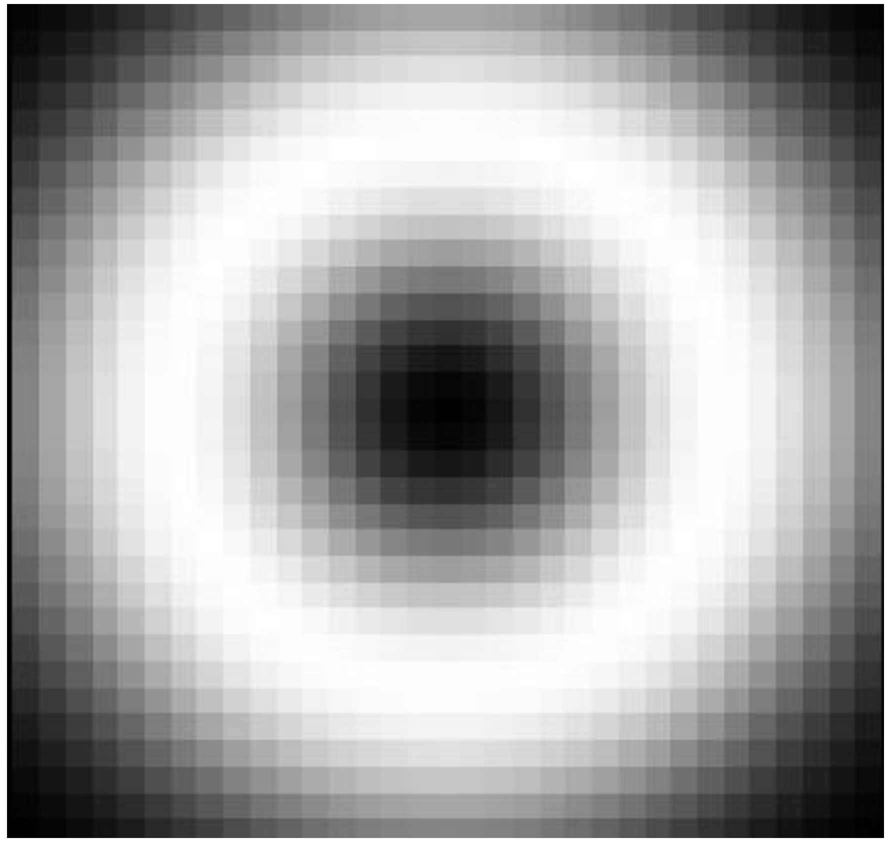


$$F(x) = e^{-ax^2}(\cos(bx^2) + i\sin(bx^2))$$

Algorithm

Separable gather (Circular Dof filter)

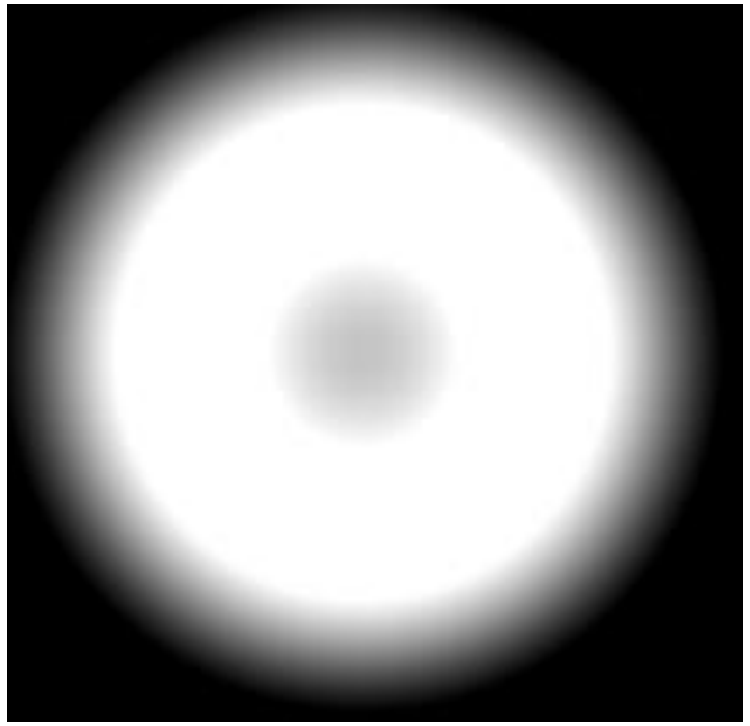




$$F(x) = e^{-ax^2}(\cos(bx^2) + i\sin(bx^2))$$

Algorithm

Separable gather (Circular Dof filter)



$$Color(x) = A * F_{real}(x) + B * F_{imaginary}(x)$$

Circular Dof: $$F(x) = e^{-ax^2}(\cos(bx^2) + i\sin(bx^2))$$

Real component

Vertical Blur

Imaginary component

Final Image

Horizontal blur & combine

Algorithm

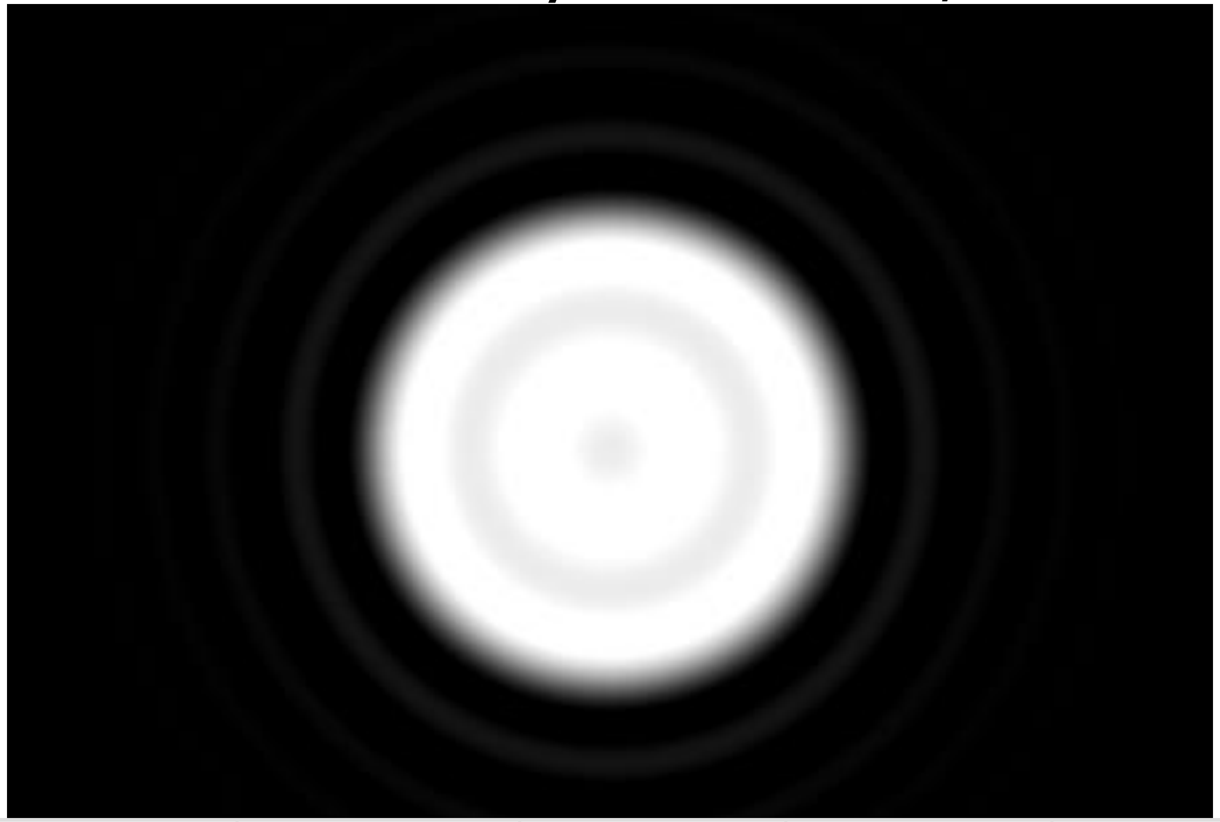
Separable gather (Circular Dof filter)

- That was just 1 component. We can add filters (multiple components) and approximate a circle better.

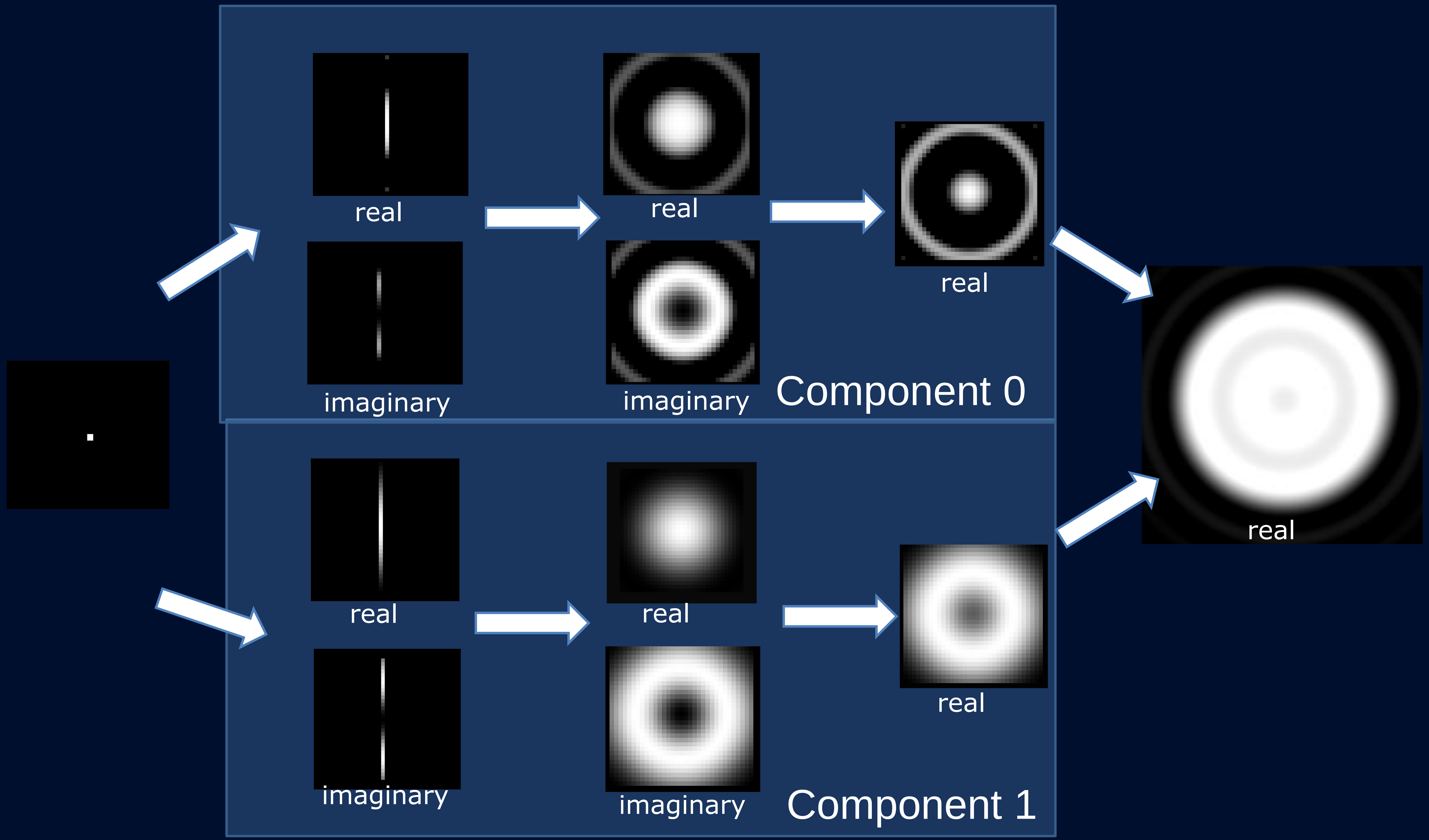
Algorithm

Two Component Filter

- We compute the filter the same way as before, but now with 2 components



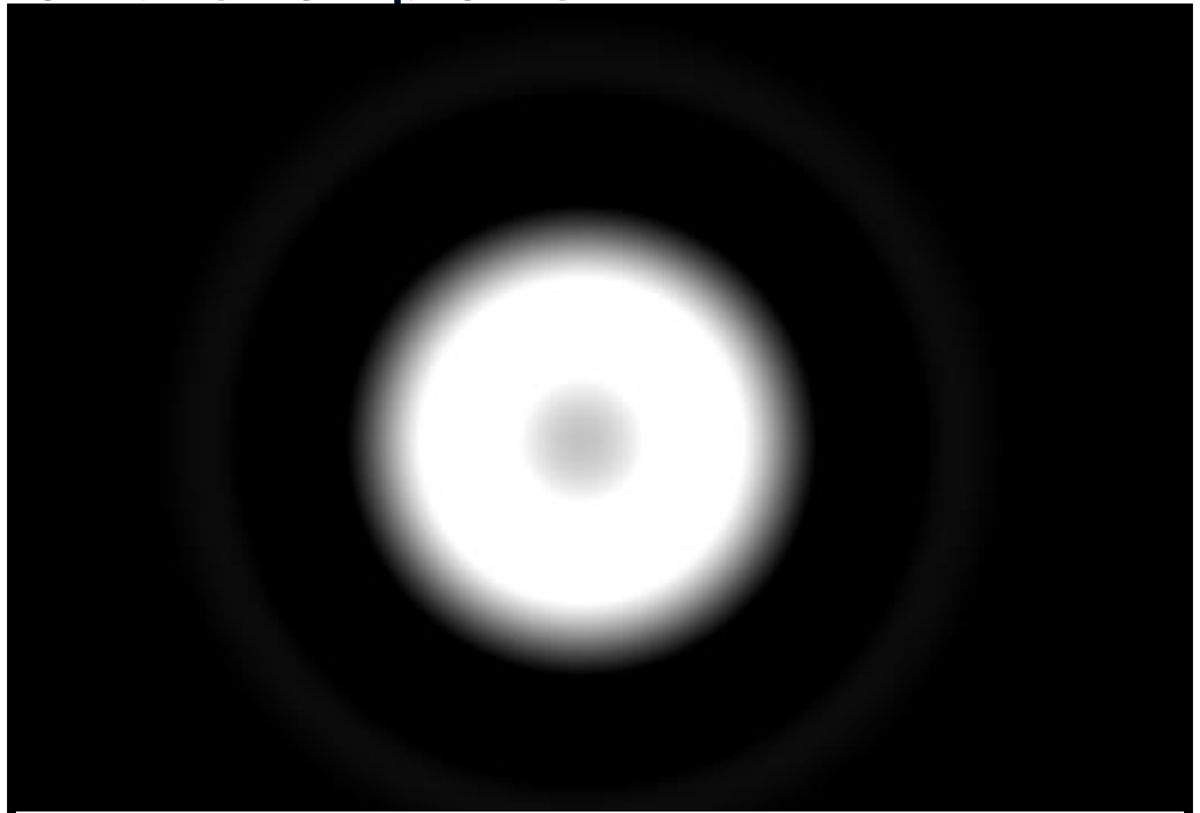
Component	a	b	A	B
C0	-0.886528	5.268909	0.411259	-0.548794
C1	-1.960518	1.558213	0.513282	4.561110


real
imaginary
real
imaginary
real
Component 0
real
imaginary
real
imaginary
real
Component 1
real

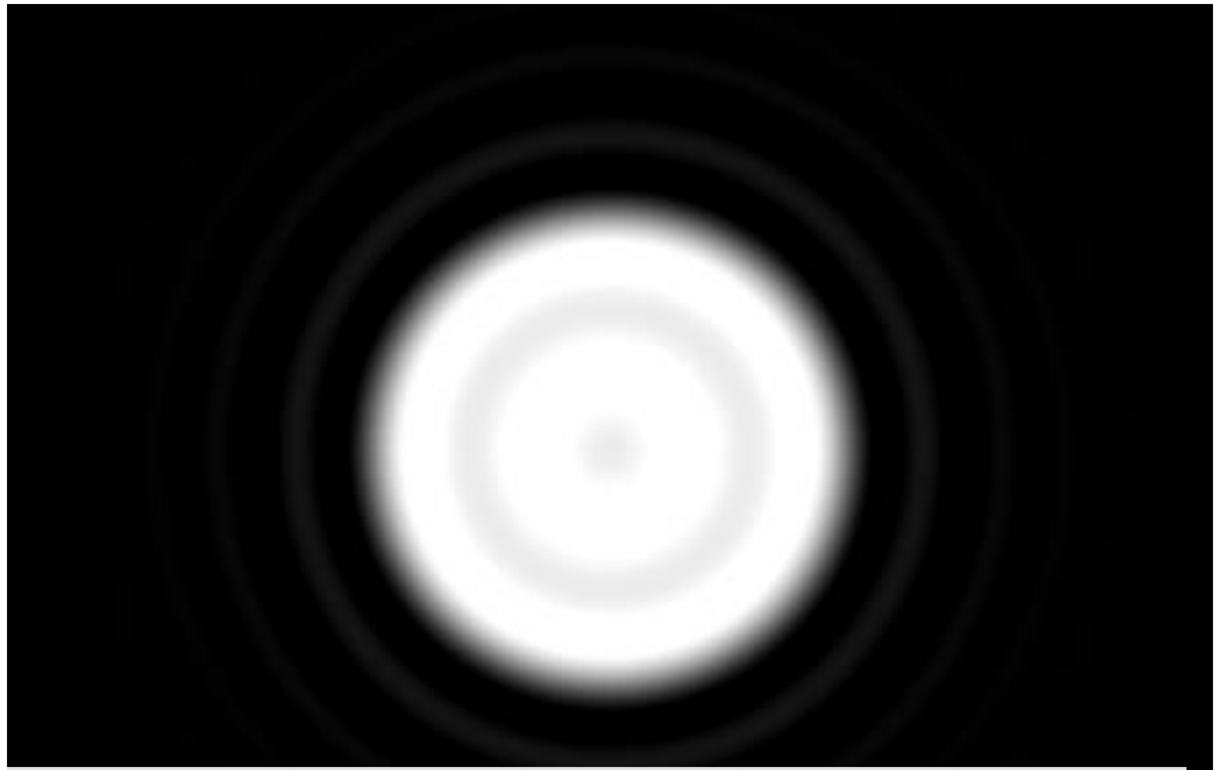
Algorithm

Circular DoF

- Low quality (1 component on left) vs High quality (2 components on the right). We use the low quality for the near blur plane, and the high quality for the far plane.



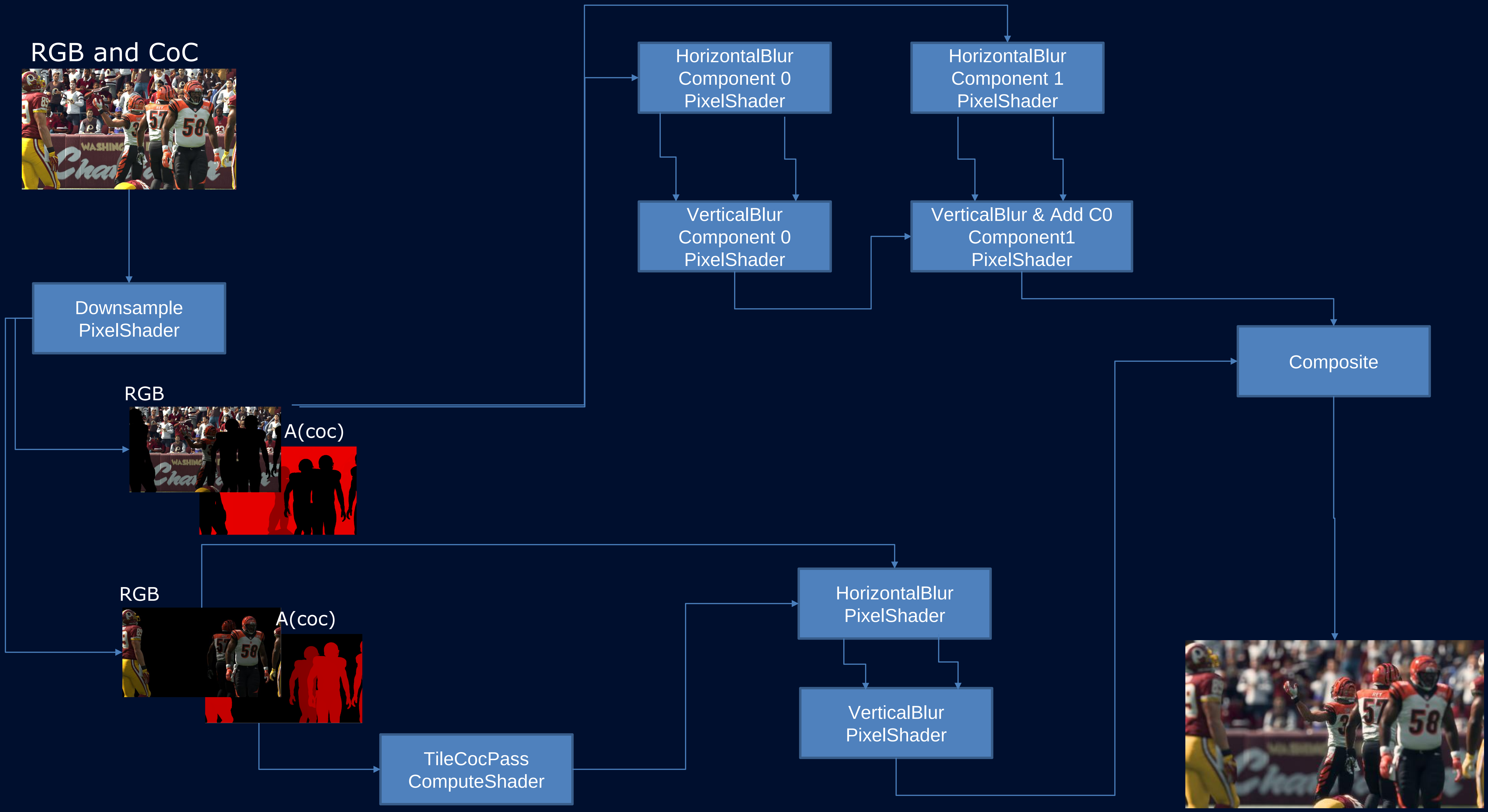
Component	a	b	A	B
C0	-0.862325	1.624835	0.767583	1.862321



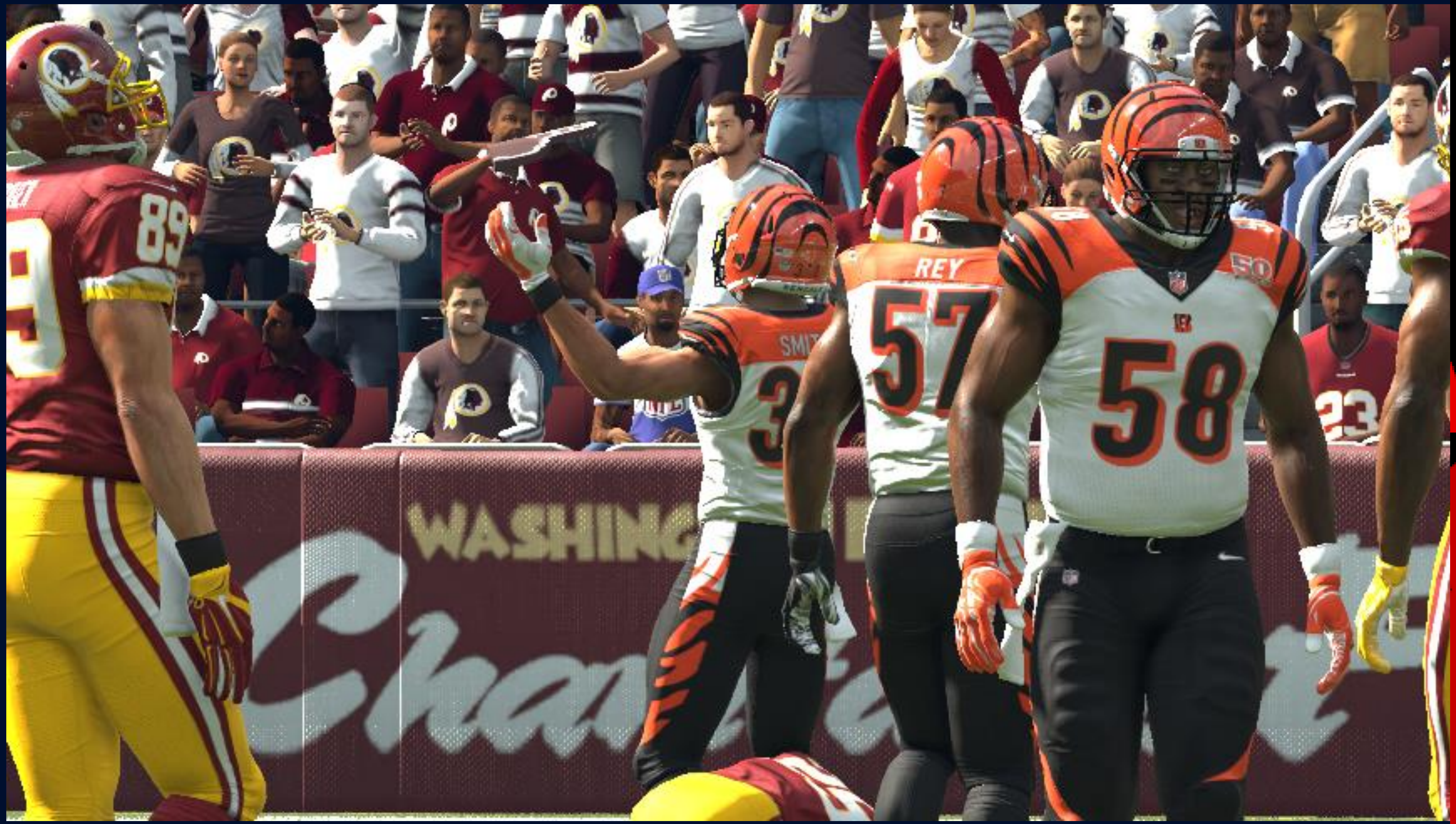
Component	a	b	A	B
C0	-0.886528	5.268909	0.411259	-0.548794
C1	-1.960518	1.558213	0.513282	4.561110

$$F(x) = e^{-ax^2}(\cos(bx^2) + i\sin(bx^2))$$
$$Color(x) = A * F_{real}(x) + B * F_{imaginary}(x)$$


REY
57
58


RGB and CoC
Downsample PixelShader
RGB
A(coc)
RGB
A(coc)
HorizontalBlur Component 0 PixelShader
HorizontalBlur Component 1 PixelShader
VerticalBlur Component 0 PixelShader
VerticalBlur & Add C0 Component1 PixelShader
Composite
HorizontalBlur PixelShader
VerticalBlur PixelShader
TileCocPass ComputeShader

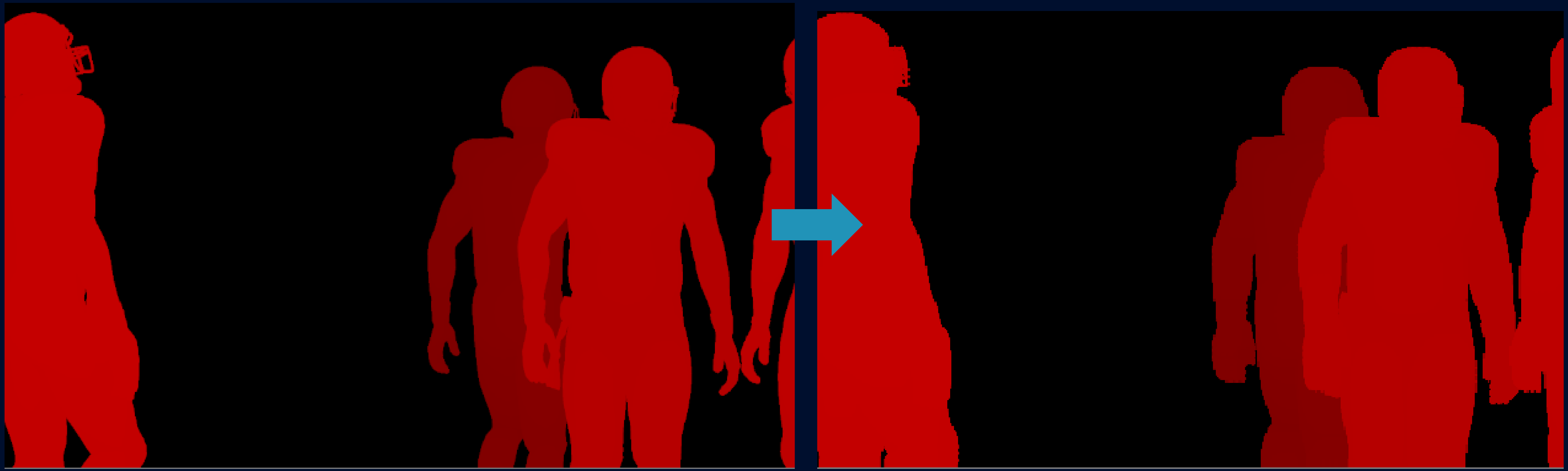
- Start with clear image and circle of confusion (near and far)

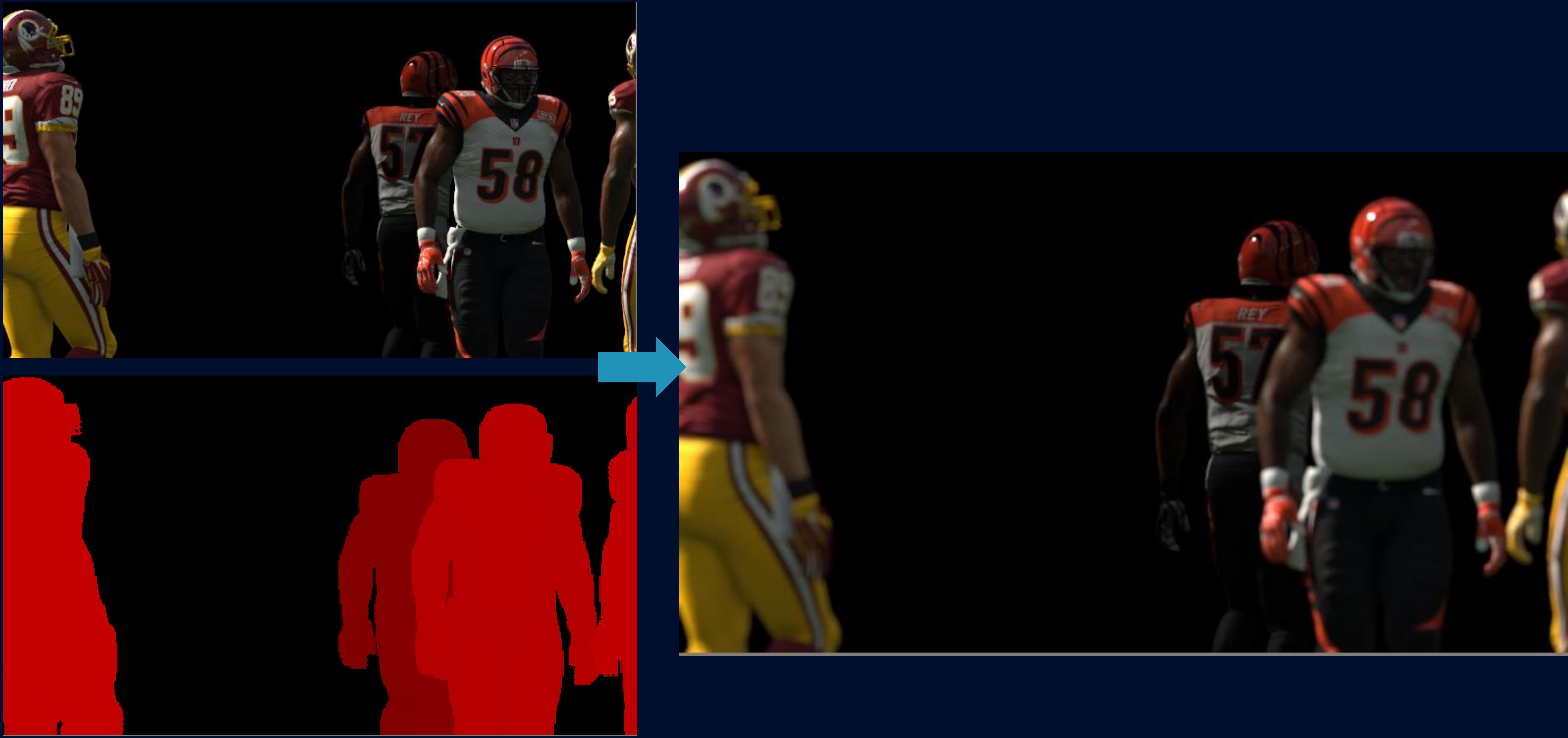


- In our case, the clear image is a 10 bit lighting buffer.
- The output would be a blurred 10 bit buffer.

- Split main image into near and far by premultiplying circle of confusion

Tile the near COC (MAX within a certain tile size to get edge bleeding)




89
REY
58


89
REY
57
58

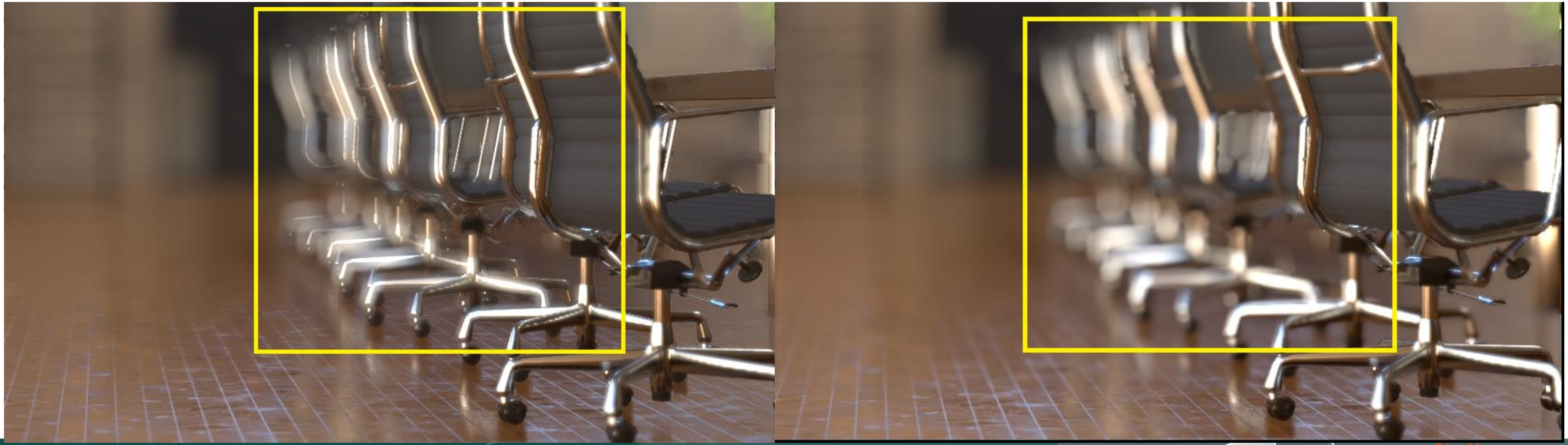

REY
57
58

Artifacts

- Occluded circles get split in 'half' due to separable nature.
- Very subtle artifact

Artifacts

- Ghosting
- Can be reduced by biasing blending (jumping to blur image as fast as possible)



Performance

- GPU (sprite dof 1080p, half res)
  - 9.98ms on XB1
  - 7.65ms on PS4
- GPU (1/4 res of 1080p, 2 components for far):
  - 1.7ms on XB1
  - 1.3ms on PS4
- GPU (1/2 res of 1080p, 1 component):
  - 3.4ms on XB1
  - 2.7ms on PS4

Performance (additional info)

- Limiting occupancy on xb1 and ps4 for downsampling pass of coc and color (full res to quarter res)
- Downsampling pass is massively vmem bound, ends up trashing the cache.
- Solution? Limit the occupancy!, can make it run as fast!
- XB1:
  - #define __XBOX_LIMIT_OCCUPANCY_WITH_HARD_LIMIT 1
- PS4:
  - #pragma argument(minvgprcount=84)
  - #pragma argument(targetoccupancy=3)
  - #pragma argument(scheduler=latency)
- special thanks to Tomasz Stachowiak [@h3r2tic]

Additional Perf Opportunities

- Explore armortization – essentially less samples are required for smaller CoC radii. We can precompute a set of filter weights, for different radius ranges, and pick them dynamically.
  - Would not improve performance on a fully blurred image.
  - Would improve performance for areas fully clear of the image.
- Combine near and far
  - Essentially have only one shader for horizontal and vertical passes
  - Use MRTs to output different values of near and far
  - Might have to explore manual occupancy hints to preserve vmem cache coherency

Additional Visual Improvements

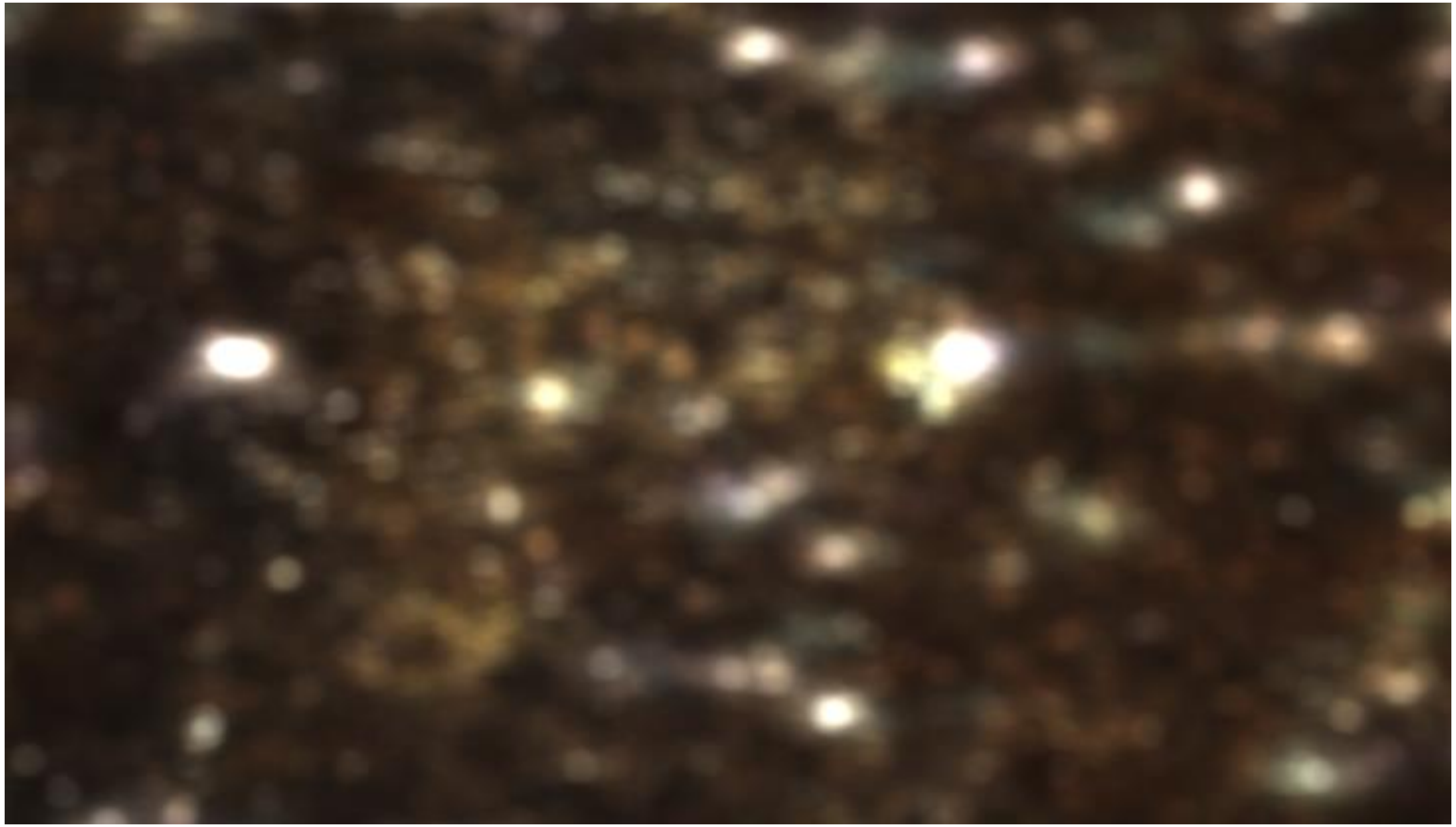
- Trasparency: instead of using transparent depth to shift COC, use multiple render planes / buckets and composite these.

- More on Ghosting
  - improved performance gains and do the pass in full resolution (see previous slide!)

  - dynamically compute pixel to ratio bias, and use scene information such as pixel luminance to automatically 'jump' to the next blur plane.

Shader Toy Example

https://www.shadertoy.com/view/Xd2BWc



PreFiltering

- Used a filter generator algorithm to precompute the filter
- Madden uses a 68 pixels (in ¼ res r = 8) diameter filter!
- It uses 2 component for far blur and 1 component for near blur.
- https://github.com/kecho/CircularDofFilterGenerator
  - A lite python version of the filter generator can be found here

Sources

- CSC algorithm blog post. (Olli Niemitalo, 2010) http://yehar.com/blog/?p=1495
- Five Rendering ideas from BF3 and NFS: e run, (Electronic Arts, Siggraph 2011) http://www.slideshare.net/DICEStudio/five-rendering-ideas-from-battlefield-3-need-for-speed-the-run

Credits

Kleber Garcia - Render Engineer, Frostbite

Karolyn Boulanger – Render Engineer, EA Sports

Arthur Homs – Principal Engineer, Microsoft

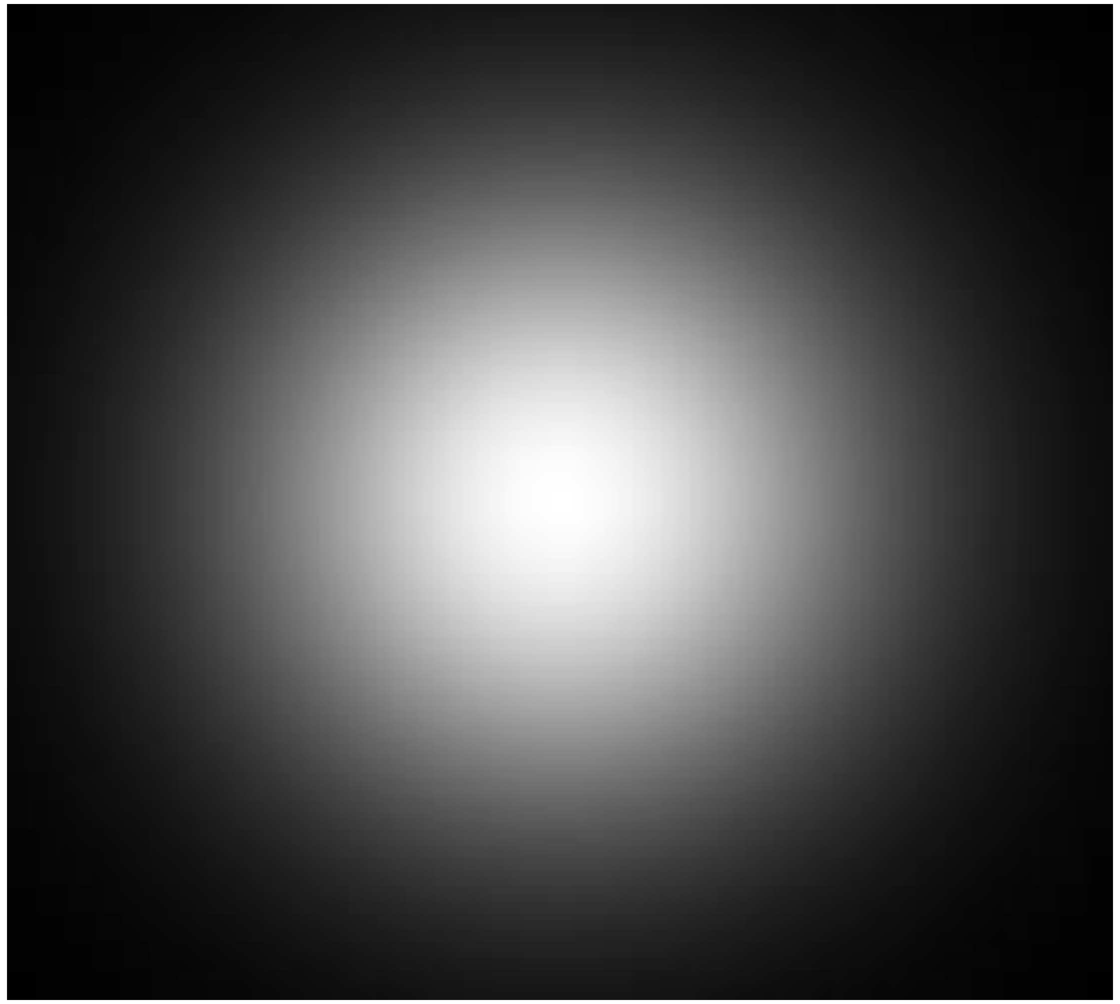
Ollie Niemitalo – Mathematician, Signal processing scientist.

Q & A

Appendix – Mathematical derivations.

F(x) filter derivation

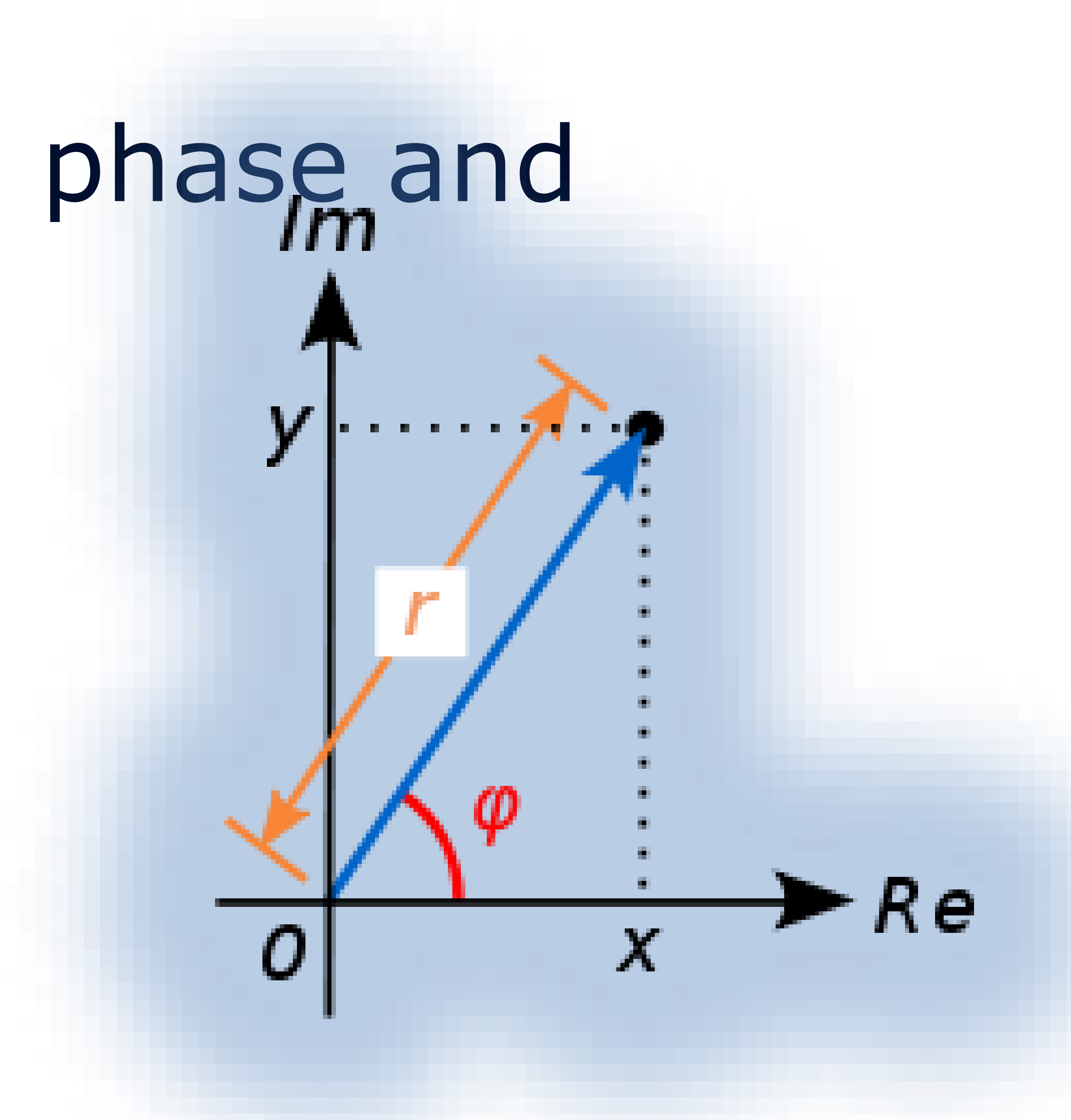
- A separable filter F(x), is separable when:
  - $F(\sqrt{(x^2+y^2)}) = F(x) * F(y)$



$F(x) = e^{-x^2}$ Gaussian function has this property! Therefore that’s why is separable

F(x) filter derivation

- An imaginary number has a phase and envelope:



- Imaginary number can be written as $x + iy$
- Or: $r(\cos\varphi + i\sin\varphi)$
- Or: $re^{i\varphi}$

F(x) filter derivation

- Let F(x) be a complex function.
- Let |F(x)| be the magnitude (r in the previous slide)
- Let arg(F(x)) be the envelope (angle *φ)*
- F(x) can be written as:
  - F(x) = | F(x) | * (cos (arg(F(x)) + i*(sin(arg(F(x))))

F(x) filter derivation

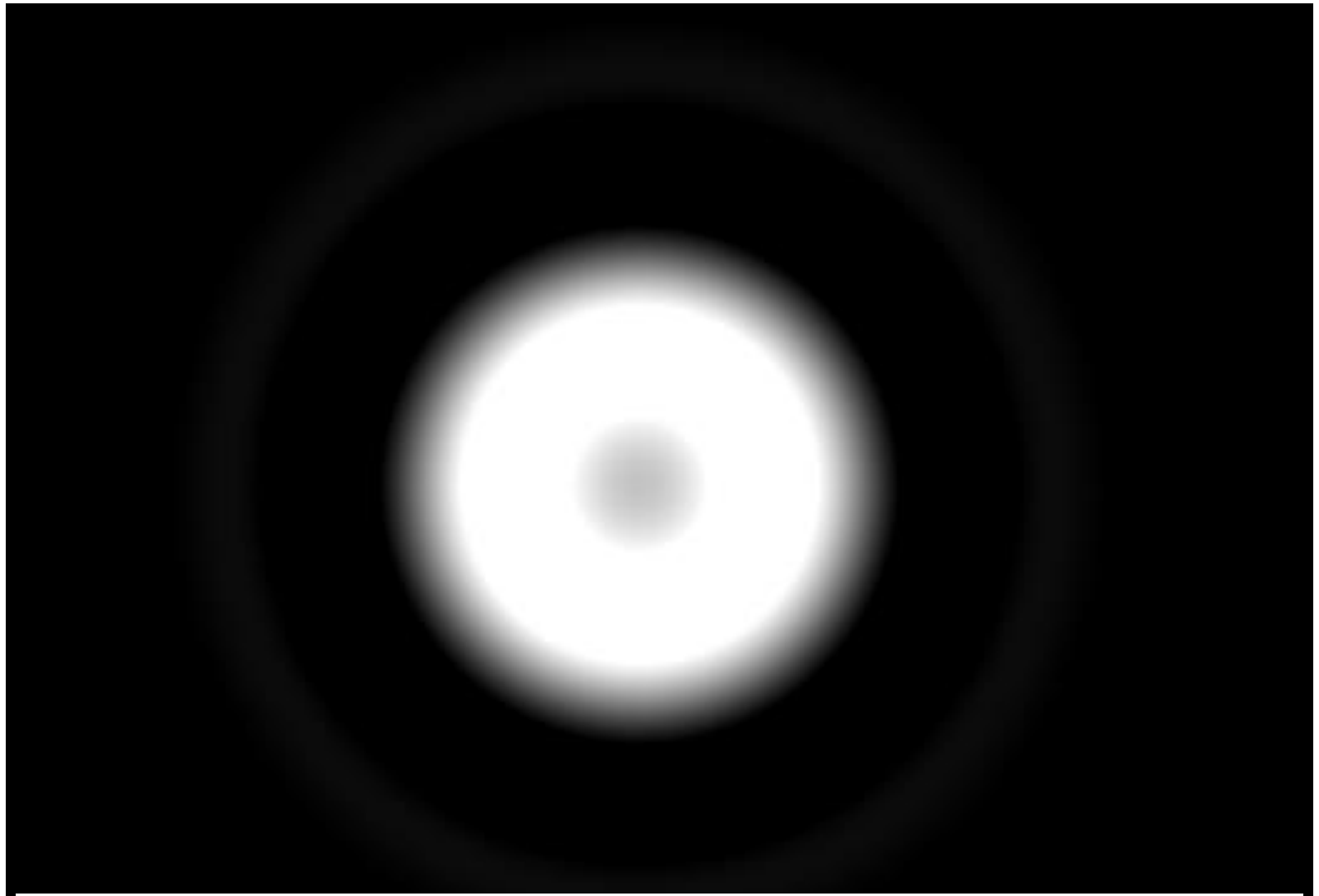
- $F(x) = | F(x) | * (\cos (\arg(F(x)) + i*(\sin(\arg(F(x))))$
  - Assume F(x) is separable.
  - Hence |F(X)| must be a Gaussian function
    - $\mathbf{|F(x)| = e^{-a*x\hat{}2}}$
  - $\arg(F(\sqrt{(x^2 + y^2)})) = \arg(F(x)) + \arg(F(y))$
    - therefore $\arg(F(x)) = \mathbf{bx^2}$

F(x) filter derivation

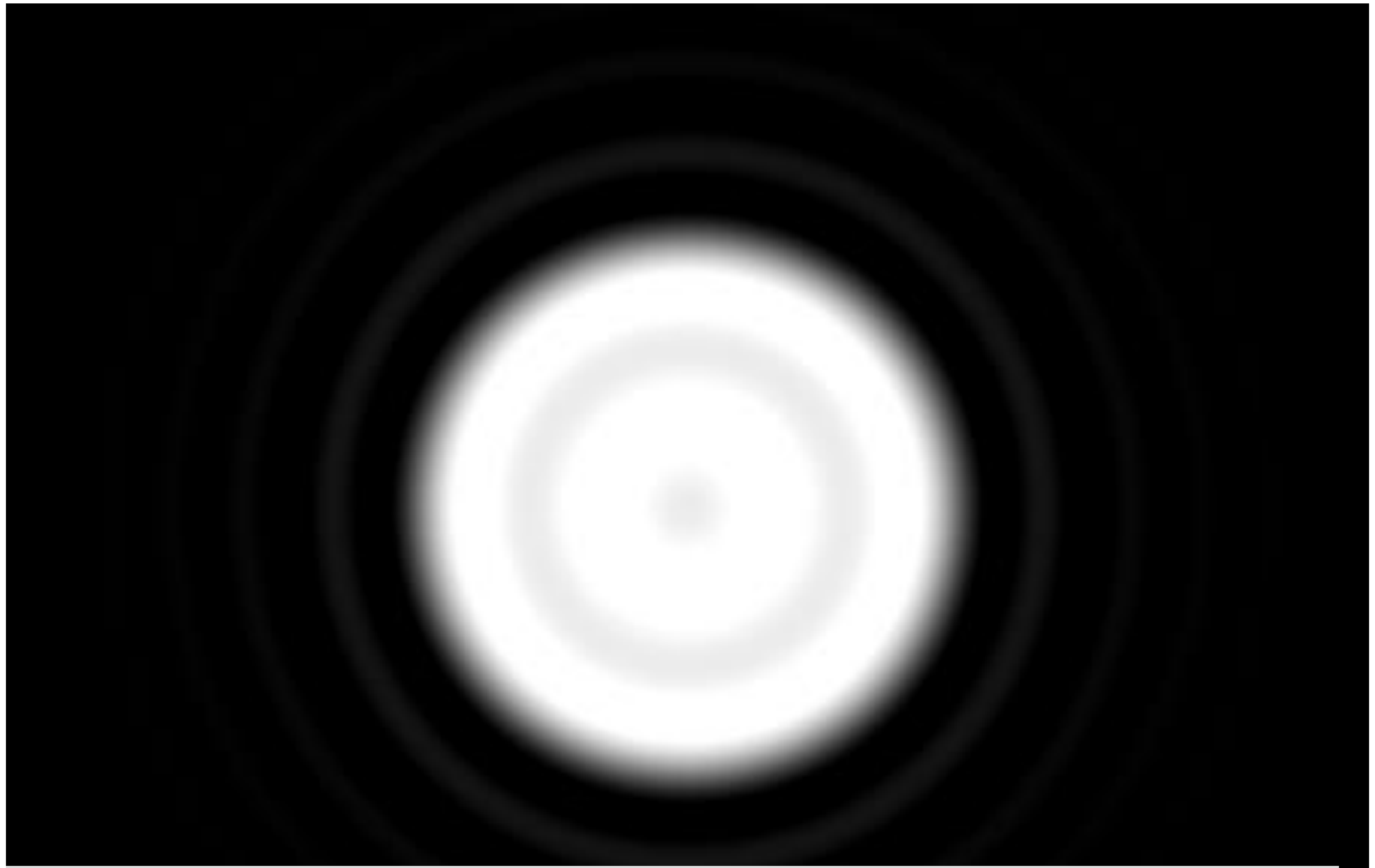
- Replacing the previous terms, we get
  - $F(x) = e^{-ax\wedge 2} * (\cos(b*x^2) + i * \sin (b*x^2))$
- Arbitrary circles can be achieved a weighted sum of imaginary and real elements.
- Final filter kernel function becomes:
  - $F_{final}(x) = A * F_{real}(x) + B * F_{imaginary}(x)$

- A final sum of these components will give us a convoluted color.

F(x) filter derivation



Component	a	b	A	B
C0	-0.862325	1.624835	0.767583	1.862321



Component	a	b	A	B
C0	-0.886528	5.268909	0.411259	-0.548794
C1	-1.960518	1.558213	0.513282	4.561110

Bracketing the filter

- How can we maximize bit precision? Bracketing and squeezing the filter to produce numbers in the [0,1] domain.

Bracketing the filter

- Let 0 < x < N, where N is the max pixel width.
- Assume we have an arbitrary G kernel with the following properties:
  - $\sum_{x=1}^{N} G(x) = V$
  - $O = Min(G(x))$
  - $\mathrm{S} = \sum_{x=1}^{N}\{G(x) - O_k\}$
- We can then transform the kernel G into the bracketed kernel G′, wich can be defined as:
  - $G'(x) = \frac{G(x) - O}{S}$
- We can then store coefficients O and S for G′(x)

Bracketing the filter

- Let I be a 1 dimensional (for simplicity) image, 16 bit rgba buffer for our final storage.
- Let I’ be a temp storage, which can only store numbers from 0 to 1 (10 bit rgba buffer)
- Let J be our initial image.
- Lets now try to convolve J using G’(x) and store it in I’[x]
  - Since we know O, and S we can store
    - I’[w] = $\sum_{x=1}^{N} J[w]G'(x)$ <- Lets instead store the bracketed version, and separately keep track of the kernel values O and S.
- I’[i] is not quite what we planned though! We want to take I’[i] and convert it to the equivalent of $\mathrm{I[w]} = \sum_{x=1}^{N} I[w]G(x)$

Bracketing the filter

- Now, we know that I' contains our bracketed filter values. When we sample from I', we can convert to the actual non bracketed by applying some inverse operations.
- We know $\mathrm{I}[\mathrm{w}] = \sum_{x=1}^{N} J[w]G(x)$
- Here is how we convert I' into I:
  - We know the definition of I'. We can expand I' algebraically into
    - $I'[w] = \sum_{x=1}^{N} J[w] * \left[\frac{G[i]-O}{S}\right]$
  - Means we can do some algebra and define I as
    - $I[w] = \sum_{x=1}^{N} \left(J[w] * \left[\frac{G[i]-O}{S}\right] * s\right) + \sum_{x=1}^{N} J[w]O$
    - $I[w] = \sum_{x=1}^{N} (I'[w] * S) + \sum_{x=1}^{N} J[w] * O$
  - Means that if we store O, S and Sum of all J[w]s (in a separate target) we can compress the render targets into 10 bits with unbounded information.