Defocus Map Estimation and Deblurring from a Single Dual-Pixel Image

ICCV 2021

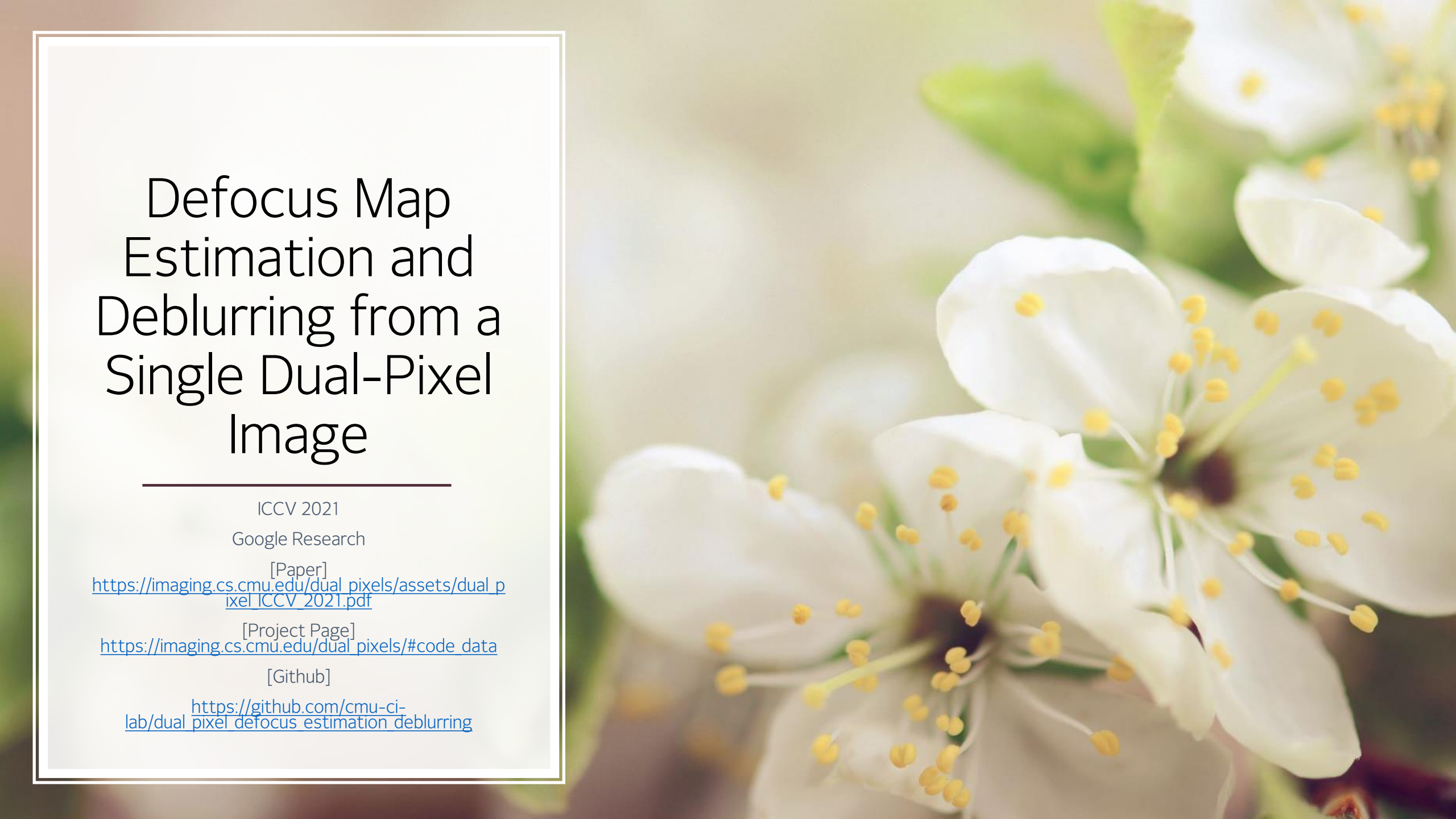
Google Research

[Paper]
https://imaging.cs.cmu.edu/dual_pixels/assets/dual_pixel_ICCV_2021.pdf

[Project Page]
https://imaging.cs.cmu.edu/dual_pixels/#code_data

[Github]

https://github.com/cmu-ci-lab/dual_pixel_defocus_estimation_deblurring



논문 소개

- 논문의 목표
  - Single Dual Pixel(DP) data 를 활용하여 밀접하게 연관된 **defocus map estimation 과 defocus blur removal 두 문제를 unsupervised 방식으로 동시에 해결**하고자 하는 것이 목표.
    - We propose a method to simultaneously recover the defocus map and all-in-focus image from a single DP capture.
    - It is beneficial to treat these two closely-connected problems simultaneously.
- 해결 방식
  - DP 영상의 광학 시스템(optics) 을 모델링하여 **optimization 기법**으로 해결.
    - We set up an optimization problem that, by carefully modeling the optics of dual-pixel images, jointly solves both problems.
  - 핵심 아이디어
    - Calibrated blur kernel 을 사용하면서 관측된 입력 DP 영상을 설명하기 위하여 **Multiplane Image(MPI) representation** 을 최적화함.
      - Then, given a single DP image, we optimize a multiplane image(MPI) representation to best explain the observed DP images using the calibrated blur kernels.
      - We use input left and right DP images **to fit a multiplane image(MPI) scene representation**, consisting of a set of fronto-parallel layers.

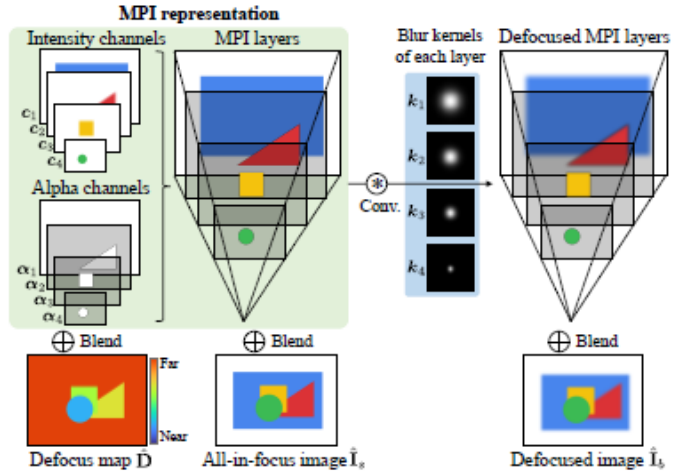


Figure 5: The multiplane image (MPI) representation consists of discrete fronto-parallel planes where each plane contains intensity data and an alpha channel. We use it to recover the defocus map, the all-in-focus image, and render a defocused image according to a given blur kernel.

MPI representation

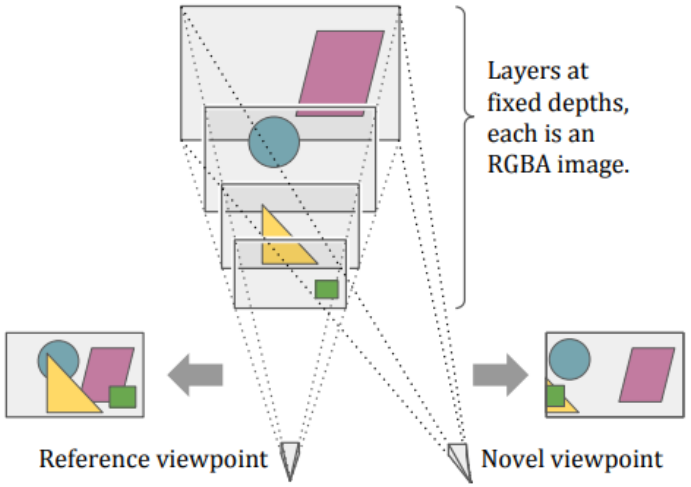
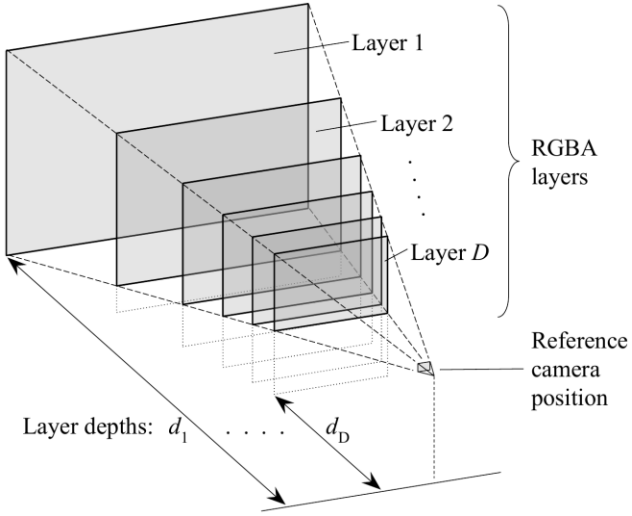


Fig. 2. An illustration of the multiplane image (MPI) representation. An MPI consists of a set of fronto-parallel planes at fixed depths from a reference camera coordinate frame, where each plane encodes an RGB image and an alpha map that capture the scene appearance at the corresponding depth. The MPI representation can be used for efficient and realistic rendering of novel views of the scene.

- MPI representation 이란
  - 영상내 장면을 특정 depth 에 대한 fronto-parallel plane 의 집합으로 표현하는 기법을 말함.
  - 이때, 각 plane 은 RGB color image 와 alpha map 으로 표현 할 수 있음.
    - The global scene representation we adopt is a **set of fronto-parallel planes** at a fixed range of depths with respect to a reference coordinate frame, where each plane d encodes an RGB color image C_d and an alpha/transparency map α_d.
- MPI representation 의 표현
  - $\{(C_1, \alpha_1), \cdots, (C_D, \alpha_D)\}$ → fronto-parallel layer
    - $where, D = number\ of\ depth.$

즉, 하나의 영상내 특정 depth 을 가지는 픽셀들의 집합이 fronto-parallel layer 이며 이러한 fronto-parallel layer 들의 집합으로 넓은 범위의 depth 를 가지는 전체 영상을 표현 할 수 있어야 함.



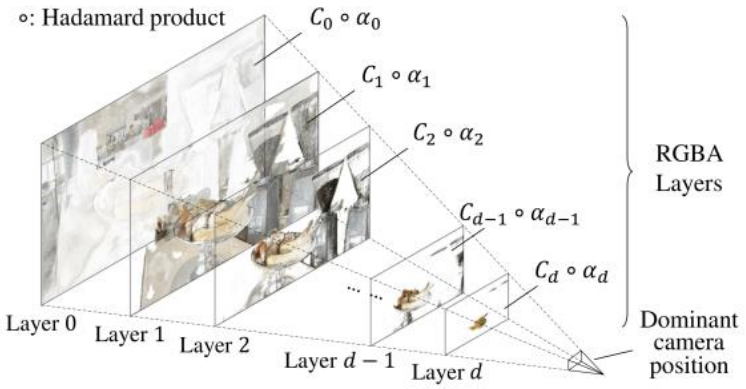
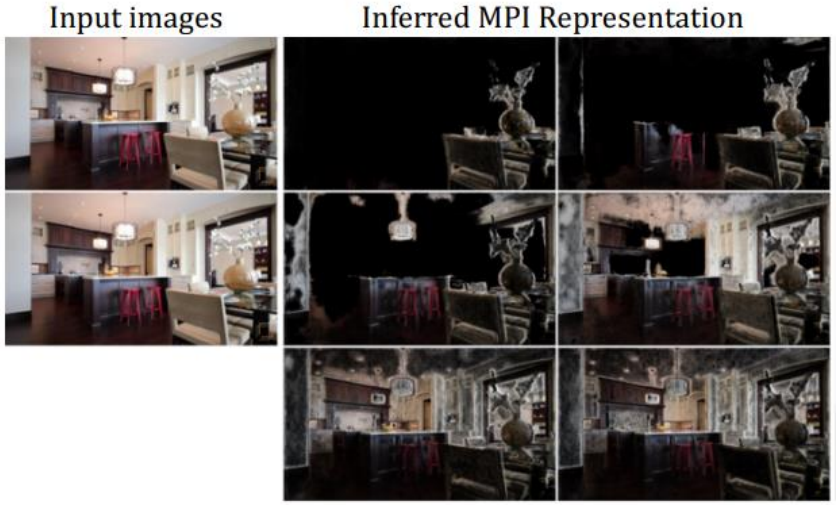


Figure 2: An illustration of MPI representation.



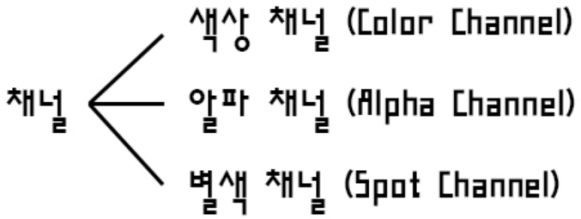
MPI representation

3.1 Multiplane image representation

The global scene representation we adopt is a set of fronto-parallel planes at a fixed range of depths with respect to a reference coordinate frame, where each plane d encodes an RGB color image C_d and an alpha/transparency map α_d. Our representation, which we call a *Multiplane Image* (MPI), can thus be described as a collection of such RGBA layers $\{(C_1, \alpha_1), \ldots, (C_D, \alpha_D)\}$, where D is the number of depth planes. An MPI is related to the *Layered Depth Image* (LDI) representation of Shade, et al. [Shade et al. 1998], but in our case the pixels in each layer are fixed at a certain depth, and we use an alpha channel per layer to encode visibility. To render from an MPI, the layers are composed from back-to-front order using the standard "over" alpha compositing operation. Figure 2 illustrates an MPI. The MPI representation is also related to the "selection-plus-color" layers used in DeepStereo [Flynn et al. 2016], as well as to the volumetric representation of Penner and Zhang [2017].

Alpha channel 의 개념

- Alpha channel 을 공부하기 전, channel 에 대한 개념을 먼저 살펴보아야 한다.
  - Channel 이란?
    - 해당 영상이 가지고 있는 색상 정보이다.
  - Channel 의 종류
    - Channel 의 종류는 크게 3종류로 색상 채널, 알파 채널, 별색 채널이 존재한다.



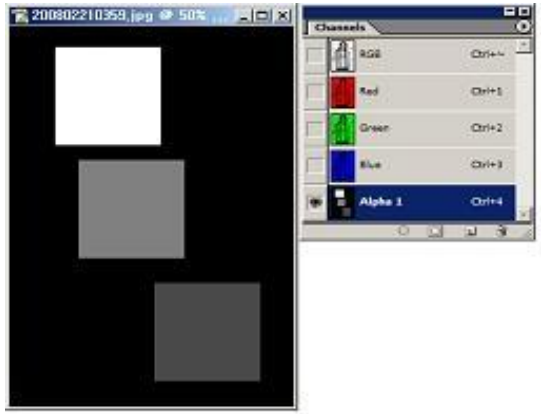
  - 색상 채널 (기본 채널)
    - **영상에 대한 기본 색상 정보를 담고 있는 channel** 로 영상을 나타낼 수 있는 기본적인 영상 표현 방식이다.
    - 대표적으로, RGB 영상이 색상 channel 의 한 종류로 각 channel 마다 Red color 에 대한 채도, Green color 에 대한 채도, Blue color 에 대한 채도 정보를 담고 있다.
- Alpha channel
  - 기본 개념
    - 영상의 편집과 합성에 도움을 주는 channel 로 **선택 영역**을 만들어 영상의 **선택 영역을 정밀하게 수정** 할 수 있는 기능을 가지고 있음.
  - 방법
    - 선택 영역을 마스트 형태로 저장하는 방식으로 이를 위하여 알파 채널을 흰색과 검은색으로 구성함.
    - **선택 영역은 흰색, 마스트 영역은 검은색** 으로 표시하며 흰색으로 된 부분은 수정 할 수 있고 검은색을 이루어진 부분은 수정 할 수 없다.
    - 뿐만 아니라, **수정되는 정도를 흰색에서 검은색 까지의 256 단계의 회색 음영을 통해서 표현** 할 수 있다. 즉, 회색 음영의 256 단계로 선택 영역의 채도에 대한 정도를 조절 하면서 RGB 색상 채널에 효과를 적용 할 수 있다.
    - 따라서, 선택영역 자체에 점진적인 효과를 줄 수 있어 영상 편집에 유용하며 화이트닝 색상 보정 등에 많이 사용된다.

http://egloos.zum.com/knyatom/v/1458411
https://m.blog.naver.com/phominator/110182845207

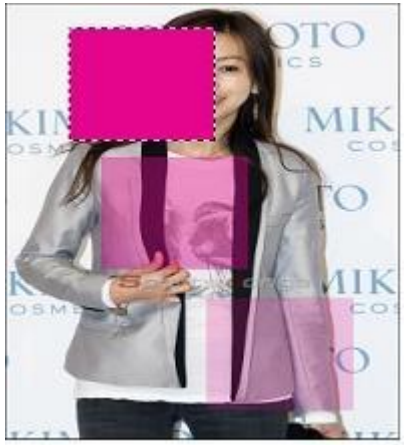
Alpha channel 의 예시

선택 영역 지정

1. RGB채널을 Ctrl + 클릭을 하면 선택영역으로 만들 수 있습니다. 채널에서는 순수한 흰색 부분이 선택영역으로 만들어 지고 순수한 검정색은 선택영역이 만들어 지지 않습니다.



2. 또한 채널의 명도에 따라 선택영역에 효과를 적용하였을 때 적용되는 범위 또한 다르게 됩니다.



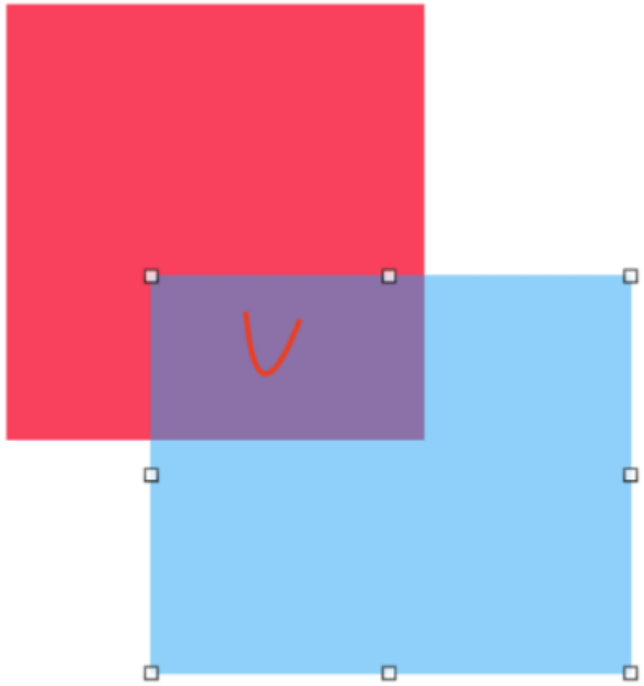
3. 완전한 흰색, 회색, 어두운 회색의 채널을 선택영역으로 만들고 자주색의 색상을 채웠다면 흰색 부분은 완전한 자주색이 채워지는 반면, 아래 검정색으로 가까이 갈수록 연하게 채워 지는 것을 알 수 있습니다.

- 즉, alpha 는 view 자체의 투명도를 나타냄.
- 0-1 사이의 값으로 0.0 은 완전하게 투명, 1.0 은 완전하게 불투명을 나타냄.

http://egloos.zum.com/knyatom/v/1458411
https://m.blog.naver.com/phominator/110182845207
https://podechannel.tistory.com/20

Opaque

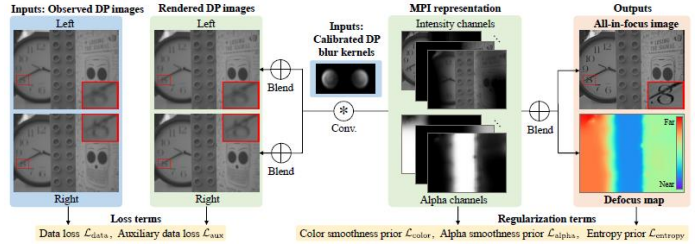
- 그렇다면, 서로 겹치는 영역이 있는 물체에 대해서는 어떻게 처리 해 주어야 할까?
  - Geometry 사 완전하게 overlap 되는 영역은 alpha 가 1 이 되며 완전하게 opaque 되었다고 말한다.
    - A value of 1 means that the pixel is fully opaque because the geometry completely overlaps the pixel window.
  - Opaque 설정이 true 로 setting 되면 view 의 그려진 부분에 대해서는 뒤에 해당하는 view 는 그려지지 않는다.



1. 빨간 네모와 파란 네모가 겹치는 부분을 drawing할 때 각 두 객체의 투명도를 계산하여 view에 표시하게 되는데 이 작업은 상당히 고비용의 작업이고 이 때 발생하는 비용을 관리하는 것이 Opaque속성이다.
2. 만약에 아래 그림에서 파란 네모의 alpha값이 1이라면, 빨간 네모와 겹쳐지는 부분을 그릴 때 겹치는 부분을 달리 표현할 필요가 없기 때문에 투명도를 불필요하게 계산할 필요가 없습니다. 따라서 이런 경우에는 Opaque를 활성화 시켜준다.
3. 즉, Opaque를 활성화시킴으로써 뒤에 있는 view와 겹치는 영역에 대해서 투명도 계산을 할 필요가 없다고 알려줌으로써 뒤에 해당하는 view 는 그리지 않는다. 따라서, drawing의 resource를 아낄 수 있게 된다.

https://yulran.tistory.com/108
https://velog.io/@yongchul/iOSUIView%EC%9D%98-drawing-%EC%86%8D%EC%84%B1Opaque-Clears-Graphics-Context
https://en.wikipedia.org/wiki/Alpha_compositing

논문 소개



- 해결 방식
  - 핵심 아이디어
    - Calibrated blur kernel 을 사용하면서 관측된 입력 DP 영상을 설명하기 위하여 **Multiplane Image(MPI) representation** 을 최적화 함.
  - 과정
    1. Left 와 right 영상의 blur kernel 을 얻기 위하여 blur kernel calibration 을 통하여 **calibrated blur kernel 을 추출**함.
       - Specifically, we perform a one-time calibration to determine the spatially-varying blur kernels for the left and right DP images.
    2. 관측된 DP 영상과 이에 1. 에서 생성한 calibrated blur kernel 을 활용하여 **multiplane image(MPI) representation 을 최적화**함.
       - Then, given a single DP image, we optimize a multiplane image(MPI) representation to best explain the observed DP images using the calibrated blur kernels.
       - multiplane image(MPI) representation **을 최적화한다는 것은 특정** defocus size, d_i **를 가지는** intensity-alpha layer **의 계수**$(\hat{C}_i, \hat{\alpha}_i)$ 를 찾는 문제와 같음.
       - 이때, DP 영상만으로 부터 MPI 를 해결하면 solution space 가 넓어 under-constraint 의 문제로 귀결됨. 이를 해결하기 위하여, 추가적인 prior 를 가하여 constrain 를 추가하도록 설계함.
         - As solving for the MPI from two DP images is under-constrained, we introduce additional priors and show their effectiveness via ablation studies.

⇒ 따라서, MPI representation 으로 부터 all-in-focus 영상과 defocus map 을 모두 생성 할 수 있기 때문에 정확한 multiplane image representation 을 최적화 하는 것이 핵심임.
- An MPI is a layered representation that accurately models occlusions, and can be used to render both defocused and all-in-focus images, as well as produce a defocus map.

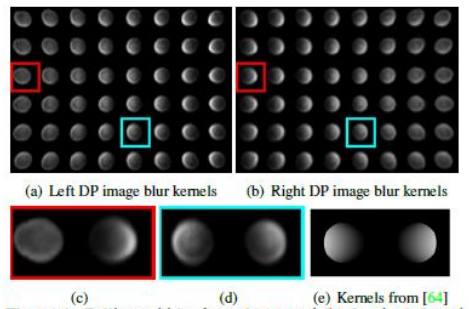


Figure 4: Calibrated blur kernels (a) and (b) for the left and right DP images. (c) and (d) show example pairs of left and right kernels marked in red and cyan. Compared to the parametric kernels (e) from [64], calibrated kernels are spatially-varying, not circular, and not left-right symmetric.

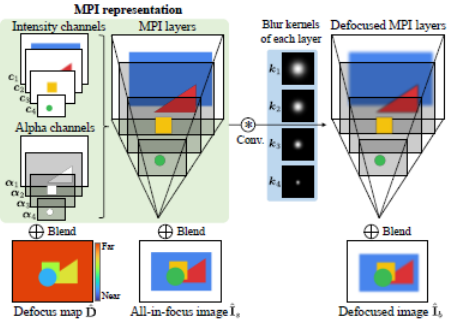


Figure 5: The multiplane image (MPI) representation consists of discrete fronto-parallel planes where each plane contains intensity data and an alpha channel. We use it to recover the defocus map, the all-in-focus image, and render a defocused image according to a given blur kernel.

논문 소개

Abstract

We present a method that takes as input a single dual-pixel image, and simultaneously estimates the image's defocus map—the amount of defocus blur at each pixel—and recovers an all-in-focus image. Our method is inspired from recent works that leverage the dual-pixel sensors available in many consumer cameras to assist with autofocus, and use them for recovery of defocus maps or all-in-focus images. These prior works have solved the two recovery problems independently of each other, and often require large labeled datasets for supervised training. By contrast, we show that it is beneficial to treat these two closely-connected problems simultaneously. To this end, we set up an optimization problem that, by carefully modeling the optics of dual-pixel images, jointly solves both problems. We use data captured with a consumer smartphone camera to demonstrate that, after a one-time calibration step, our approach improves upon prior works for both defocus map estimation and blur removal, despite being entirely unsupervised.

We propose a method to simultaneously recover the defocus map and all-in-focus image from a single DP capture. Specifically, we perform a one-time calibration to determine the spatially-varying blur kernels for the left and right DP images. Then, given a single DP image, we optimize a *multiplane image* (MPI) representation [77, 91] to best explain the observed DP images using the calibrated blur kernels. An MPI is a layered representation that accurately models occlusions, and can be used to render both defocused and all-in-focus images, as well as produce a defocus map. As solving for the MPI from two DP images is under-constrained, we introduce additional priors and show their effectiveness via ablation studies. Further, we show that in the presence of image noise, standard optimization has a bias towards underestimating the amount of defocus blur, and we introduce a bias correction term. Our method does not require large amounts of training data, save for a one-time calibration, and outperforms prior art on both defocus map estimation and blur removal, when tested on images captured using a consumer smartphone camera. We make our implementation and data publicly available [85].

기존 방식 및 기존 방식의 문제점

- (monocular) Depth estimation
  1. Single image 로 depth 를 예측하는 방식
     - Depth-dependent disparity cues 를 활용 할 수 없기 때문에 고유의 모호성(inherent ambiguity) 이 존재한다는 문제가 있음.
       - Despite these advances, producing high quality depth from a single image remains difficult, due to the inherent ambiguities of monocular depth estimation.
  2. Single DP image 로 depth 를 예측하는 방식
     - 기존 방식들은 DP 영상으로 부터 depth 를 예측하기 위하여 무수히 많은 DP 영상들과 정답 depth map 을 수집해야 한다는 단점이 있음.
       - trained neural networks to output depth from DP images, using a captured dataset of thousands of DP images and ground truth depth maps [3]. The resulting performance improvements come at a significant data collection cost.
- Defocus deblurring
  - DPD Net
    - DSLR 카메라의 aperture size 를 달리하여 정답 in-focus image 와 입력 defocus image 를 수집하여 이를 supervised 방식으로 학습하는 방식임.
      - Wide aperture
        - 렌즈 구멍을 크게 하여 빛을 많이 받아들이는 대신 DpF(Depth of Focus) 가 좁기 때문에 초점이 맞지 않는 영역이 생김.
      - Narrow aperture
        - 렌즈 구멍을 작게 하여 빛을 많이 받아들이는 대신 DoF(Depth of Focus) 가 넓기 때문에 초점이 맞는 영역이 대부분임.
    - 하지만, smart phone camera 는 aperture 를 조절 할 수 없는 경우가 대부분으로 정답 DP 영상을 수집하기 어려우며 DSLR 카메라에 촬영된 영상으로 학습된 DPD Net 은 smart phone camera 영상에는 일반화 되기 어렵다는 단점이 있음.
      - Their method relies on a dataset of pairs of wide and narrow aperture images captured with a DSLR, and may not generalize to images captured on smartphone cameras, which have very different optical characteristics.
      - Such a dataset is impossible to collect on smartphone cameras with fixed aperture lenses.

⟹ 따라서, 본 방식은 정답 DP 영상의 필요 없이 DP 영상의 defocus blur 특성을 활용하여 정교한 모델링을 통해서 defocus map 과 all-in-focus 영상을 동시에 예측하는 모델을 제안함.

기존 방식 및 기존 방식의 문제점

2. Related Work

Depth estimation. Multi-view depth estimation is a well-posed and extensively studied problem [30, 71]. By contrast, single-view, or *monocular*, depth estimation is ill-posed. Early techniques attempting to recover depth from a single image typically relied on additional cues, such as silhouettes, shading, texture, vanishing points, or data-driven supervision [5, 7, 10, 13, 29, 37, 38, 42, 44, 51, 67, 70, 72]. The use of deep neural networks trained on large RGBD datasets [17, 22, 50, 52, 69, 74] significantly improved the performance of data-driven approaches, motivating approaches that use synthetic data [4, 28, 56, 60, 92], self-supervised training [23, 25, 26, 39, 54, 90], or multiple data sources [18, 66]. Despite these advances, producing high-quality depth from a single image remains difficult, due to the inherent ambiguities of monocular depth estimation.

Recent works have shown that DP data can improve monocular depth quality, by resolving some of these ambiguities. Wadhwa *et al.* [82] applied classical stereo matching methods to DP views to compute depth. Punnappurath *et al.* [64] showed that explicitly modeling defocus blur during stereo matching can improve depth quality. However, they assume that the defocus blur is spatially invariant and symmetric between the left and right DP images, which is not true in real smartphone cameras. Depth estimation with DP images has also been used as part of reflection removal algorithms [65]. Garg *et al.* [24] and Zhang *et al.* [87] trained neural networks to output depth from DP images, using a captured dataset of thousands of DP images and ground truth depth maps [3]. The resulting performance improvements come at a significant data collection cost.

Focus or defocus has been used as a cue for monocular depth estimation prior to these DP works. Depth from defocus techniques [19, 63, 78, 84] use two differently-focused images with the same viewpoint, whereas depth from focus techniques use a dense focal stack [27, 33, 76]. Other monocular depth estimation techniques use defocus cues as supervision for training depth estimation networks [75], use a coded aperture to estimate depth from one [46, 81, 89] or two captures [88], or estimate a defocus map using synthetic data [45]. Lastly, some binocular stereo approaches also explicitly account for defocus blur [12, 49]; compared to depth estimation from DP images, these approaches assume different focus distances for the two views.

Defocus deblurring. Besides depth estimation, measuring and removing defocus blur is often desirable to produce sharp all-in-focus images. Defocus deblurring techniques usually estimate either a depth map or an equivalent defocus map as a first processing stage [14, 40, 62, 73]. Some techniques modify the camera hardware to facilitate this stage. Examples include inserting patterned occluders in the camera aperture to make defocus scale selection easier [46, 81, 89, 88]; or sweeping through multiple focal settings within the exposure to make defocus blur spatially uniform [59]. Once a defocus map is available, a second deblurring stage employs non-blind deconvolution methods [46, 21, 43, 83, 57, 86] to remove the defocus blur.

Deep learning has been successfully used for defocus deblurring as well. Lee *et al.* [45] train neural networks to regress to defocus maps, that are then used to deblur. Abuolaim and Brown [1] extend this approach to DP data, and train a neural network to directly regress from DP images to all-in-focus images. Their method relies on a dataset of pairs of wide and narrow aperture images captured with a DSLR, and may not generalize to images captured on smartphone cameras, which have very different optical characteristics. Such a dataset is impossible to collect on smartphone cameras with fixed aperture lenses. In contrast to these prior works, our method does not require difficult-to-capture large datasets. Instead, it uses an accurate model of the defocus blur characteristics of DP data, and simultaneously solves for a defocus map and an all-in-focus image.

Dual-Pixel Image Formation

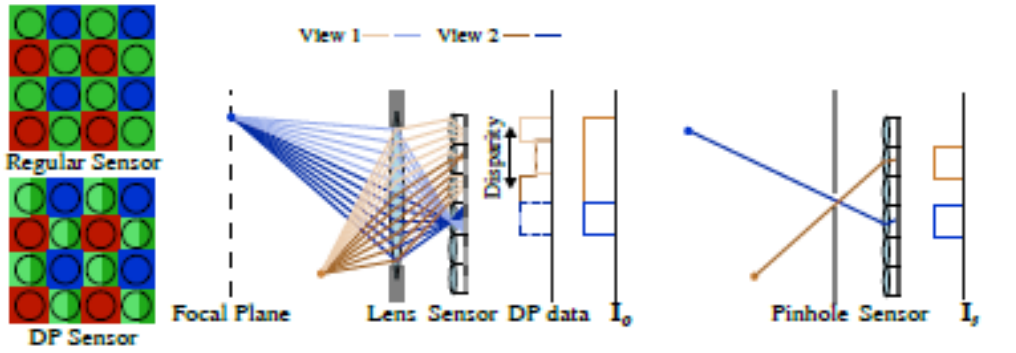


Figure 3: A regular sensor and a DP sensor (a) where each green pixel is split into two halves. For a finite aperture lens (b), an in-focus scene point produces overlapping DP images, whereas an out-of-focus point produces shifted DP images. Adding the two DP images yields the image that would have been captured by a regular sensor. (c) shows the corresponding pinhole camera where all scene content is in focus. Ignoring occlusions, images in (b) can be generated from the image in (c) by applying a depth-dependent blur.

Kernel 의 모양은 카메라의 aperture 의 모양임.

- All focus(c) 영상에 depth dependent blur 를 적용하여 DP 영상(b)을 생성 할 수 있다.
  - 이를 구체화 하면 다음과 같다.
    - Thin lens model 에서, blur 영상은 sharp 영상을 depth-dependent kernel 으로 blur 시키는 것과 같음. 이때, depth-dependent kernel 의 모양은 크기가 d 인 aperture 의 모양과 같으며 보통 반지름이 d 인 circular disk 로 근사함.
      - Under the thin lens model, the blurred image I_0 of the out-of-focus point equals blurring I_s with a depth-dependent kernel k_d, shaped as d-scaled version of the aperture(typically circular disk of radius d).

$$d = A + {}^{B}/_{Z}$$

      - Point depth, d and lens dependent constant, A, B
    - 따라서, 반지름 d per pixel kernel 들의 집합인 defocus map, D 는 inverse depth, ${}^{1}/_{Z}$ 의 linear function 으로 depth map 의 근사치임.
  - 이때, DP 영상의 disparity 가 defocus blur size 와 비례관계에 있기 때문에 DP 영상의 disparity 단서가 defocus map estimation 과 defocus blur removal 에 중요한 단서임.

Dual-Pixel Image Formation

3. Dual-Pixel Image Formation

We begin by describing image formation for a regular and a dual-pixel (DP) sensor, to relate the defocus map and the all-in-focus image to the captured image. For this, we consider a camera imaging a scene with two points, only one of which is in focus (Fig. 3(b)). Rays emanating from the in-focus point (blue) converge on a single pixel, creating a sharp image. By contrast, rays from the out-of-focus point (brown) fail to converge, creating a blurred image.

If we consider a lens with an infinitesimally-small aperture (i.e., a pinhole camera), only rays that pass through its center strike the sensor, and create a sharp all-in-focus image $\mathbf{I}_s$ (Fig. 3(c)). Under the thin lens model, the blurred image $\mathbf{I}_o$ of the out-of-focus point equals blurring $\mathbf{I}_s$ with a depth-dependent kernel k_d, shaped as a d-scaled version of the aperture—typically a circular disc of radius $d = A + {}^{B}/_{Z}$, where Z is the point depth, and A and B are lens-dependent constants [24]. Therefore, the per-pixel signed kernel radius d, termed the *defocus map* $\mathbf{D}$, is a linear function of inverse depth, thus a proxy for the depth map. Given the defocus map $\mathbf{D}$, and ignoring occlusions, the sharp image $\mathbf{I}_s$ can be recovered from the captured image $\mathbf{I}_o$ using non-blind deconvolution. In practice, recovering either the defocus map $\mathbf{D}$ or the sharp image $\mathbf{I}_s$ from a single image $\mathbf{I}_o$ is ill-posed, as multiple $(\mathbf{I}_s, \mathbf{D})$ combinations produce the same image $\mathbf{I}_o$. Even when the defocus map $\mathbf{D}$ is known, determining the sharp image $\mathbf{I}_s$ is still ill-posed, as blurring irreversibly removes image frequencies.

DP sensors make it easier to estimate the defocus map. In DP sensors (Fig. 3(a)), each pixel is split into two halves, each collecting light from the corresponding half of the lens aperture (Fig. 3(b)). Adding the two half-pixel, or DP, images $\mathbf{I}_o^l$ and $\mathbf{I}_o^r$ produces an image equivalent to that captured by a regular sensor, i.e., $\mathbf{I}_o = \mathbf{I}_o^l + \mathbf{I}_o^r$. Furthermore, DP images are identical for an in-focus scene point, and shifted versions of each other for an out-of-focus point. The amount of shift, termed *DP disparity*, is proportional to the blur size, and thus provides an alternative for defocus map estimation. In addition to facilitating the estimation of the defocus map $\mathbf{D}$, having two DP images instead of a single image provides additional constraints for recovering the underlying sharp image $\mathbf{I}_s$. Utilizing these constraints requires knowing the blur kernel shapes for the two DP images.

Blur kernel calibration

- 모델의 입력으로 DP 영상 뿐만 아니라, calibrated left and right blur kernel 이 필요함.
- 그렇다면, calibrated blur kernel 은 어떻게 만들까?
  - Mannan and Langer 논문 [1] 을 방식을 착용함.

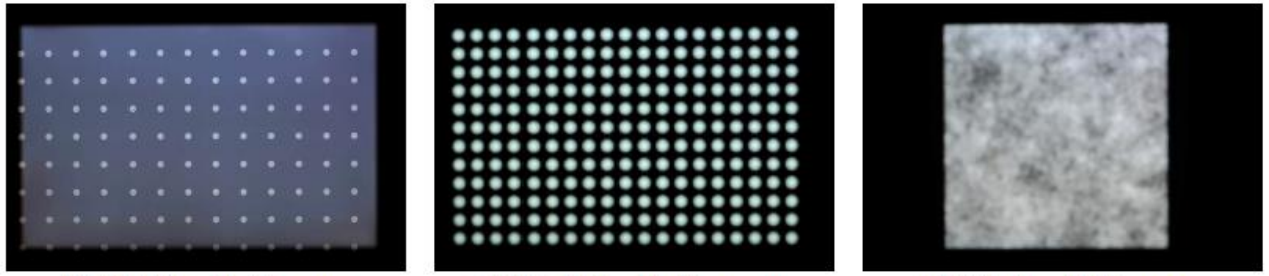
2016 13th Conference on Computer and Robot Vision

Blur Calibration for Depth from Defocus

Fahim Mannan* and Michael S. Langer†
School of Computer Science
McGill University
Montreal, Quebec H3A 0E9, Canada
{*fmannan, †langer}@cim.mcgill.ca

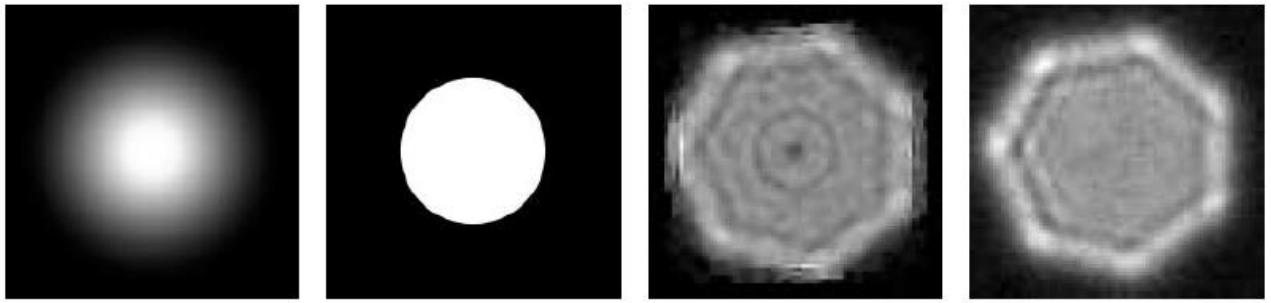
***Abstract*—Depth from defocus based methods rely on measuring the depth dependent blur at each pixel of the image. A core component in the defocus blur estimation process is the depth variant blur kernel. This blur kernel is often approximated as a Gaussian or pillbox kernel which only works well for small amount of blur. In general the blur kernel depends on the shape of the aperture and can vary a lot with depth. For more accurate blur estimation it is necessary to precisely model the blur kernel. In this paper we present a simple and accurate approach for performing blur kernel calibration for depth from defocus. We also show how to estimate the relative blur kernel from a pair of defocused blur kernels. Our proposed approach can estimate blurs ranging from small (single pixel) to sufficiently large (e.g. 77×77 in our experiments). We also experimentally demonstrate that our relative blur estimation method can recover blur kernels for complex asymmetric coded apertures which has not been shown before.**

***Keywords*-Depth from Defocus, Point spread functions, Relative Blur, Optimization**



(a) Grid of Dots with 10s exposure (b) Grid of Disks with 0.2s exposure (c) $1/f$ texture with 0.5s exposure

Figure 2: (a) and (b) are defocused images of the calibration patterns. We use the disk images for PSF estimation and the dot images for qualitative comparison. (c) defocused test image used for evaluating the DFD models. For all these images the object to sensor distance is 1.5 m and focus distance is 0.5 m. The captured images are of size 4288×2848 pixels but we only use the center part for our experiments.

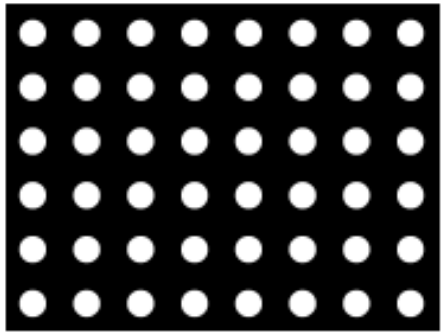


(a) Observed blurred disk image (b) Estimated sharp disk (c) Estimated PSF from (a) and (b) (d) Observed pixel (green channel)

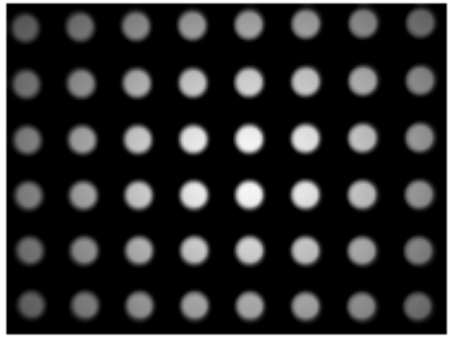
Figure 3: Example of PSF estimation from observed blurred disk image. (a) Observed disk (199×199 pixels), (b) estimated sharp disk image based on the projected size of the disk grid, (c) PSF estimated using Eq. 5, and (d) image of a single pixel (green channel). Object to sensor distance is 1.5 m and focus distance 0.5 m, and f-number $f/11$. The size of the PSF kernel is 77×77. Note that diffraction effects (e.g.ringing, brighter corners, etc.) as well as the aperture-stop's shape are also captured in the estimated PSF.

Blur kernel calibration 방법

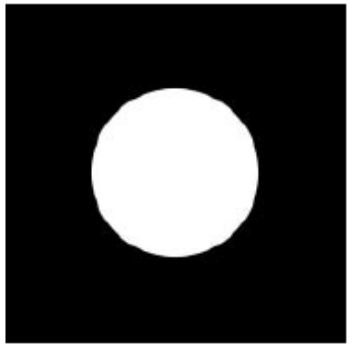
- 기본 아이디어
  1. 카메라에서 특정 거리가 떨어진 모니터 화면에 regular gird 내 원형 디스크들을 띄어 놓고 이를 촬영함.



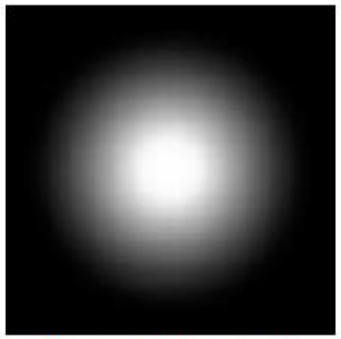
(b) Calibration pattern



(a) Captured image



(b) Estimated sharp disk



(a) Observed blurred disk image

  2. 이때, 촬영된 영상은 모니터 영상 화면의 영상 즉, latent sharp image 에서 aperture shape 에 따라 blur 가 발생한 관측 영상이 출력됨. 따라서, 우리는 latent image, i_S 와 observed blurred image, i_B 를 둘 다 얻을 수 있음.
     - 이때, 실제로는 촬영된 영상에 대하여 thresholding 과 binarize 과정이 필요하며 latent sharp image 을 생성하고 align 하는 과정이 추가됨.
  3. Aligned latent image 와 observed image 를 통해서 [1] 에서 제안된 **Quadratic Programming(QR) 문제**를 풀어 spatially varying blur kernel 을 얻어 낼 수 있음.

$$argmin_h \lambda_j \left\| f_j * (i_S * h - i_B) \right\|_2^2 + \lambda_{n+1} \|\nabla h\|_2^2 + \lambda_{n+2} \|R^\circ h\|_2^2$$
$$subject\ to\ \|h\|_1 = 1, h \geq 0$$

- i_S = sharp image, i_B = observed blurred image, h = blur kernel that is to be estimated.
- f_j = filter that is applied to the image.
- R = spatial regularization matrix which in this case is a parabola to ensure that the kernel goes to zero near the edge.

Blur kernel calibration 방법

IV. PSF and Relative Blur Estimation

A. Blur PSF Estimation

After radiometric correction of the calibration image, 25 disk patches are extracted from the center of the image and averaged. Then the latent sharp disk image is created based on the projected disk center distance. The absolute PSF is estimated by taking a sharp and a blur image pair and solving the following Quadratic Programming (QP) problem.

$$\begin{aligned} \underset{h}{\operatorname{argmin}} & \sum_{j=1}^{n} \lambda_j \|f_j * (i_S * h - i_B)\|_2^2 \\ & + \lambda_{n+1}\|\nabla h\|_2^2 + \lambda_{n+2}\|R \circ h\|^2 \qquad (5) \\ \text{subject to } & \|h\|_1 = 1\,,\ h \geq 0. \end{aligned}$$

In the above optimization problem, i_B is the observed blurred image and i_S is the sharp image. h is the PSF kernel that is to be estimated. f_j is a filter that is applied to the images. In the experiments, we use $f_1 = \delta$, $f_2 = G_x$, and $f_3 = G_y$, where G_* is the spatial derivative of a Gaussian in the horizontal and vertical directions. The matrix R – in the element-wise product with the kernel h – is a spatial regularization matrix which in this case is a parabola to ensure that the kernel goes to zero near the edge. The constraints ensure that the kernel is non-negative and preserves the mean intensity after convolution. The optimization function is similar to the one proposed by Ens and Lawrence except in this case we formulate the problem in 2D and in the filter space with explicit non-negativity and unity constraints. The convolution operation and derivative of the kernel operators can be expressed using a convolution matrix [17] and the optimization problem can be solved using off-the-shelf QP solvers (in our case Matlab's `quadprog`).

Blur kernel calibration in DP data

1. [1] 에서 제안한 방법과 유사하게 left 와 right 영상의 calibrated blur kernel 을 독립적으로 생성함.
2. 추가적으로 left 와 right 영상의 calibrated vignetting 을 생성함.
   - 방식은 diffuser 를 통해 white sheet 에 대한 6개의 영상을 촬영한 후, 이를 left 와 right 영상에 대하여 각각 평균을 구하여 vignetting pattern, W_l 과 W_r 을 얻어냄.
     - Specifically, for the same focus distance as above, we capture six images of a white sheet through a diffuser. We then average all left and right images individually to obtain the left and right vignetting patterns W_l and W_r, respectively.

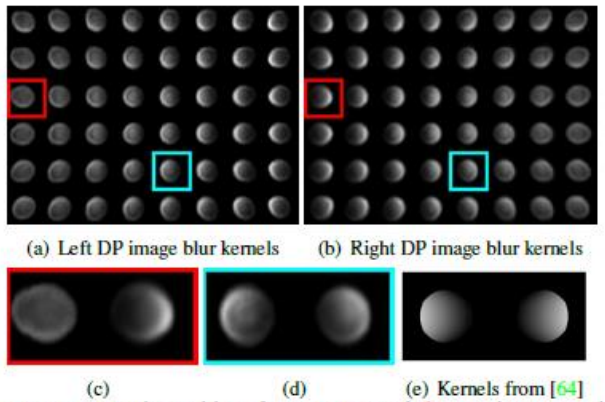


Figure 4: Calibrated blur kernels (a) and (b) for the left and right DP images. (c) and (d) show example pairs of left and right kernels marked in red and cyan. Compared to the parametric kernels (e) from [64], calibrated kernels are spatially-varying, not circular, and not left-right symmetric.

[2] 논문에서 가정한 parametric kernel 과 달리 blur calibration 통해서 얻어진 calibrated kernel 은 완벽한 원형도 아니며 왼쪽과 오른쪽이 완벽한 대칭 관계도 아니다.

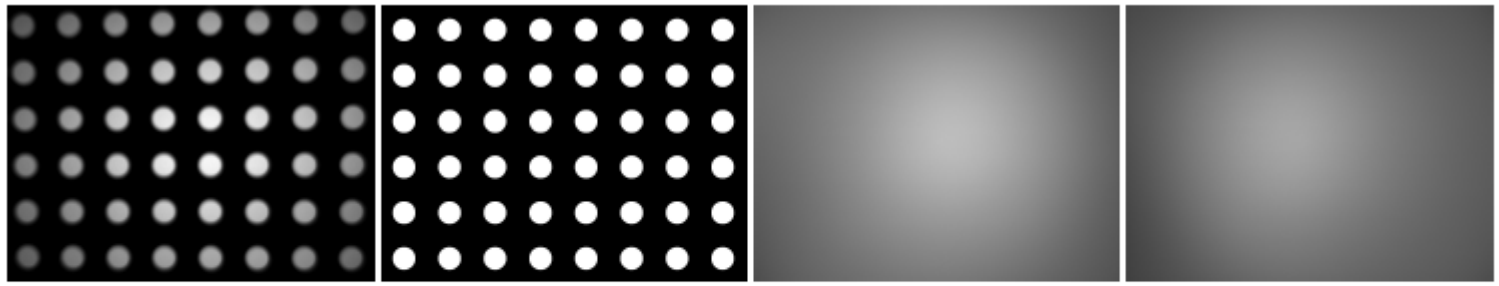


Figure 1: Captured image (a) of the calibration pattern (b) that is used to calibrate the blur kernels. Left DP image (c) and right DP image (d) of a white sheet shot through a diffuser that is used to correct for vignetting.

바네팅(vignetting) 현상

- 바네팅 현상이란?
  - 그림과 같이 주변부의 광량 저하로 렌즈 중심을 지나는 광축 선 부근이 가장 밝고 영상의 모서리 혹은 외곽 부분으로 갈수록 어두워 지거나 검게 가려지는 현상을 말함.

그림. 바네팅 현상의 예시 사진

- 바네팅 현상의 원인
  - 바네팅 현상의 원인은 그림과 같이 렌즈 내부에서 광선이 센서까지 도달하지 못하기 때문에 발생함.

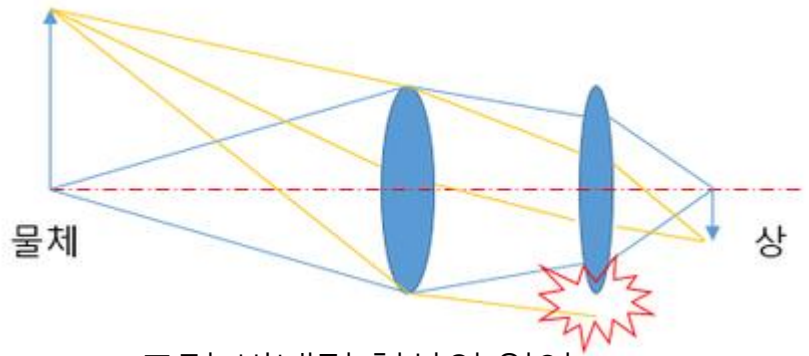


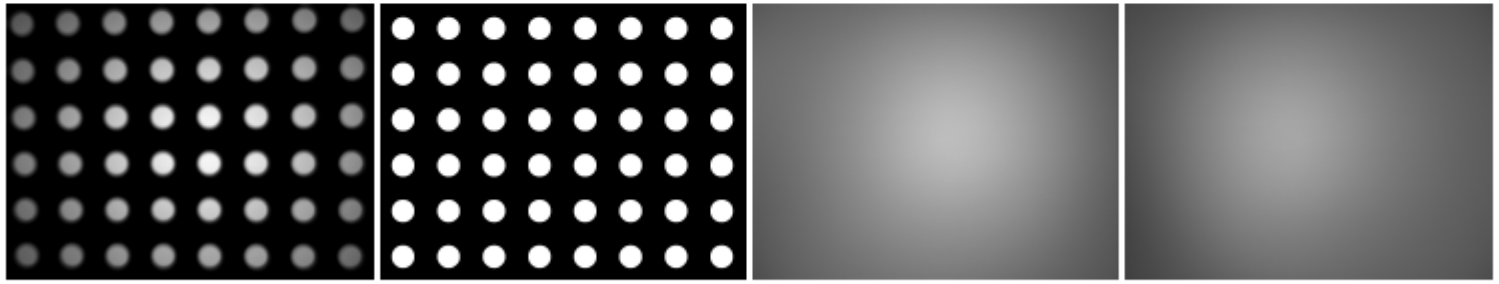
그림. 바네팅 현상의 원인

https://visionblog.vieworks.com/knowledge/optic/lens-evaluation-relative-illumination/

Blur kernel calibration in DP data

2. Blur Kernel Calibration

We provide more information about our calibration procedure for the left and right blur kernels used as input to our method. We use a method similar to the one proposed by Mannan and Langer [6], and calibrate blur kernels for left and right dual-pixel (DP) images independently (Fig. 1) for a specific focus distance. Specifically, we image a regular grid of circular discs on a monitor screen at a distance of $\sim 45\,\mathrm{cm}$ from the camera. We apply global thresholding and binarize the captured image, perform connected component analysis to identify the individual discs and their centers, and generate and align the binary sharp image M with the known calibration pattern by solving for a homography between the calibration target disc centers and the detected centers. In order to apply radiometric correction, we also capture all-white and all-black images displayed on the same screen, and generate the grayscale latent sharp image as $I_l = M \odot I_w + (1 - M) \odot I_b$, where $\odot$ represents pixel-wise multiplication, and I_w and I_b are captured all-white and all-black images. Once we have the aligned latent image and the captured image, we can solve for spatially-varying blur kernels using the optimization proposed by Mannan and Langer [6]. Specifically, we solve for a 8×6 grid of kernels corresponding to 1344×1008 central field of view.



(a) Captured image (b) Calibration pattern (c) Left vignetting (d) Right vignetting

Figure 1: Captured image (a) of the calibration pattern (b) that is used to calibrate the blur kernels. Left DP image (c) and right DP image (d) of a white sheet shot through a diffuser that is used to correct for vignetting.

In addition to the blur kernels, we calibrate for different vignetting in left and right DP images. Specifically, for the same focus distance as above, we capture six images of a white sheet through a diffuser. We then average all left and right images individually to obtain the left and right vignetting patterns W_l and W_r, respectively.

Blur kernel calibration. As real lenses have spatially-varying kernels, we calibrate an 8×6 grid of kernels. To do this, we fix the focus distance, capture a regular grid of circular discs on a monitor screen, and solve for blur kernels for left and right images independently using a method similar to Mannan and Langer [55]. When solving for kernels, we assume that they are normalized to sum to one, and calibrate separately for vignetting: we average left and right images from six captures of a white diffuser, using the same focus distance as above, to produce left and right vignetting patterns W_l and W_r. We refer to the supplement for details.

We show the calibrated blur kernels in Fig. 4. We note that these kernels deviate significantly from parametric models derived by extending the thin lens model to DP image formation [64]. In particular, the calibrated kernels are spatially-varying, not circular, and not symmetric.

Blur kernel calibration in DP data

Note: To run the code on your own Google Pixel 4 data, please adjust the preprocessing step in `./code/util.py/load_data_and_calibration` if needed, such that the input dual pixel images are normalized to the range of [0, 1].

```python
# Load input DP data & calibrated blur kernels
patch_params = dict(patch_size=168, num_rows=6, num_cols=8)
observations, blur_kernels = util.load_data_and_calibration(input_params['data_dir'], input_params['dp_file'], patch_params)
# Normalize input DP data
observations_max = np.max(observations)
observations /= observations_max
```

```python
def load_data_and_calibration(data_dir, dp_file, patch_params):
  """Load Google Pixel 4 DP data and calibration

  Args:
    data_dir: data directory
    dp_file: dual-pixel filename
    patch_params: a dictionary containing image patch parameters

  Returns:
    left and right dp images: [2, H, W, C]
    calibrated blur kernels: [#rows * #cols, F, F], F: blur kernel size
  """

  def _load_and_preprocess_pixel_data(path_to_file):
    # first deduct black level (1024 for 14-bit Google Pixel 4 DP data), then normalize to [0, 1]
    with PIL.Image.open(path_to_file) as f:
      image = onp.array(f) - 1024
      image[image < 0] = 0
      image = np.stack([np.float32(image)] * 1, axis=2) / (2 ** 14 - 1)
    return image
  left_image = _load_and_preprocess_pixel_data(os.path.join(data_dir, f'{dp_file}_left.png'))
  right_image = _load_and_preprocess_pixel_data(os.path.join(data_dir, f'{dp_file}_right.png'))


  calibration_dir = 'calibration'
  # Vignetting correction
  left_white_calib = _load_and_preprocess_pixel_data(os.path.join(data_dir, calibration_dir, f'white_sheet_left.png'))
  right_white_calib = _load_and_preprocess_pixel_data(os.path.join(data_dir, calibration_dir, f'white_sheet_right.png'))
  per_pixel_scale = left_white_calib / right_white_calib
  window_size = 101
  per_pixel_scale_avg = flax.nn.avg_pool(per_pixel_scale[None], (window_size, window_size), strides=(1, 1), padding='SAME')[0]
  right_image = per_pixel_scale_avg * right_image
```

```python
# Keep only central field of view (1008 * 1344)
def _crop_image_central_fov(images, patch_size, num_rows, num_cols):
  """ Crop images

  Args:
    images: [..., H, W, C] #images, height, width, #channels.

  Returns:
    [..., #rows * P, #cols * P, C] cropped images
  """

  crop_y = patch_size * num_rows
  crop_x = patch_size * num_cols
  offset_y = (images.shape[-3] - crop_y) // 2
  offset_x = (images.shape[-2] - crop_x) // 2

  return images[..., offset_y:offset_y + crop_y, offset_x:offset_x + crop_x, :]
observations = np.stack([left_image, right_image], axis=0)
observations = _crop_image_central_fov(observations, **patch_params)

# Calibrated blur kernels
blur_kernels_left = np.load(os.path.join(data_dir, calibration_dir, f'blur_kernels_left.npy'))
blur_kernels_right = np.load(os.path.join(data_dir, calibration_dir, f'blur_kernels_right.npy'))

return observations, np.stack([blur_kernels_left, blur_kernels_right], axis=1)
```

그림. White sheet left

그림. White sheet right

Proposed method

- Problem formulation
  - 입력
    - 1. 두개의 single channel DP 영상(gray scale 만 입력으로 받을 수 있다는 말 같음), I_o^l, I_o^r 과 2. calibrated left and right blur kernels, $k_d^l, k_d^r \; with \; certain \; defocus \; size \; d$ 가 입력됨.
      - 이때, defocus size d' 은 defocus size d 의 blur kernel 에 대하여 $^{d'}/_{d}$ 만큼 scaling 을 통하여 구할 수 있음. 즉, defocus blur kernel 들은 서로 shape 은 일치하고 size 만 변화함.
  - 출력
    - 관측된 입력 데이터를 가장 잘 설명 할 수 있는 multiplane image(MPI) representation 을 최적화하여 이를 가지고 1. all-in-focus 영상, $\hat{I}_S$ 와 2. defocus map, D 를 출력함.

```
# Compute scaled blur kernels
scales = np.linspace(mpi_end_scale, mpi_start_scale, num=num_mpi_layers, endpoint=True)
optim_params['scales'] = scales
filter_halfwidth = blur_kernels.shape[-1] // 2
blur_kernels_scaled = util.rescale_blur_kernels(blur_kernels, filter_halfwidth*2+1, scales)

precomputed_vars = \
  dict(observations = observations,
       observations_volume = np.repeat(observations[None, ...], repeats=num_mpi_layers, axis=0),
       filter_halfwidth = filter_halfwidth,
       blur_kernels_scaled = blur_kernels_scaled,
       bias_correction = util.compute_bias_correction(observations, blur_kernels_scaled) )
```

```
# Initialize color channels of all MPI layers to be the mean of the input DP images,
#           alpha channels to be [1, 1/2, 1/3, ..., 1/num_of_mpi_layers] s.t. all layers have the same transmittance
mpi_colors = onp.ones((num_mpi_layers, *observations.shape[1:])) * np.mean(observations)
mpi_alphas = onp.ones((*mpi_colors.shape[:-1], 1)) / onp.arange(1., num_mpi_layers + 1., 1.)[..., None, None, None]
mpi_init = onp.concatenate([mpi_colors, mpi_alphas], axis=-1)

print(' ---> Start optimization ...')
outputs = optim.optimization([logit(mpi_init)], precomputed_vars, optim_params, patch_params)
```

4. Proposed Method

The inputs to our method are two single-channel DP images, and calibrated left and right blur kernels. We correct for vignetting using W_l and W_r, and denote the two vignetting-corrected DP images as $\mathbf{I}_o^l$ and $\mathbf{I}_o^r$, and their corresponding blur kernels at a certain defocus size d as k_d^l and k_d^r, respectively. We assume that blur kernels at a defocus size d' can be obtained by scaling by a factor $^{d'}/_{d}$ [64, 88]. Our goal is to optimize for the multiplane image (MPI) representation that best explains the observed data, and use it to recover the latent all-in-focus image $\hat{\mathbf{I}}_s$ and defocus map $\hat{\mathbf{D}}$. We first introduce the MPI representation, and show how to render defocused images from it. We then formulate an MPI optimization problem, and detail its loss function.

Optimizer parameters for reproducing results in the paper

```python
def set_up_parameters(i_file=0):
  """The following parameters are for reproducing results in the paper:

  Shumian Xin, Neal Wadhwa, Tianfan Xue, Jonathan T. Barron, Pratul P. Srinivasan, Jianwen Chen, Ioannis Gkioulekas, and Rahul Garg.
  "Defocus Map Estimation and Deblurring from a Single Dual-Pixel Image", ICCV 2021.

  Args:
    i_file: DP file index

  Returns:
    num_of_files: total number of files in the data directory
    input_params: a dictionary of input DP info (directory & filename)
    optim_params: a dictionary of optimization parameters
  """

  data_dir = '../DP_data_pixel_4'
  num_of_files = 17

  # Input parameters
  input_params = \
    dict(data_dir = data_dir,
         dp_file=f'{i_file + 1:03d}')

  # Optimization parameters
  num_mpi_layers = 12
  mpi_start_scales = [0.5, 0.5, 0.7, 0.7, 0.1, 0.1, 0.1, 0.2, 0.1, 0.2, 0.2, 0.1, 0.3, 0.1, 0.1, 0.3, 0.2]
  mpi_end_scales = [1.6, 1.6, 1.6, 1.6, 1.3, 1.4, 1.4, 1.5, 1.5, 1.5, 1.5, 1.5, 1.6, 1.5, 1.6, 1.5, 1.6]
  weights_prior_sharp_im_tv = \
    [30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0]
  weights_prior_alpha_tv = \
    [1.5e4, 1.5e4, 1.5e4, 1.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4, 7.5e4]
  weights_prior_entropy = \
    [20.0, 20.0, 20.0, 20.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 30.0, 20.0, 20.0, 20.0, 20.0, 20.0]

  optim_params = \
    dict(init_lr = 3e-1,
         final_lr = 1e-1,
         num_iters = 10000,
         weight_loss_data = 2.5e4,
         weight_loss_aux_data = 2.5e4,
         weight_prior_sharp_im_tv = weights_prior_sharp_im_tv[i_file],
         weight_prior_alpha_tv = weights_prior_alpha_tv[i_file] * ((num_mpi_layers / 12) ** 2),
         weight_prior_entropy = weights_prior_entropy[i_file] / (-np.log(np.sum((np.ones((num_mpi_layers,)) / num_mpi_layers) ** 2))),
         num_mpi_layers = num_mpi_layers,
         mpi_start_scale = mpi_start_scales[i_file],
         mpi_end_scale = mpi_end_scales[i_file])

  return num_of_files, input_params, optim_params
```

Code 실행

Code

Code implementation is in `./code`. It is written in Python, with autograd package Jax. **Note:** When installing Jax, make sure to install with GPU support.

To reproduce results in the paper (in the paper, results are postprocessed for better visualization), run:

```
cd ./code; python ./run.py
```

Each optimization runs for 10,000 iterations with an Adam optimizer, and takes about 2 hours on an Nvidia Titan RTX GPU.

Code has been tested with:

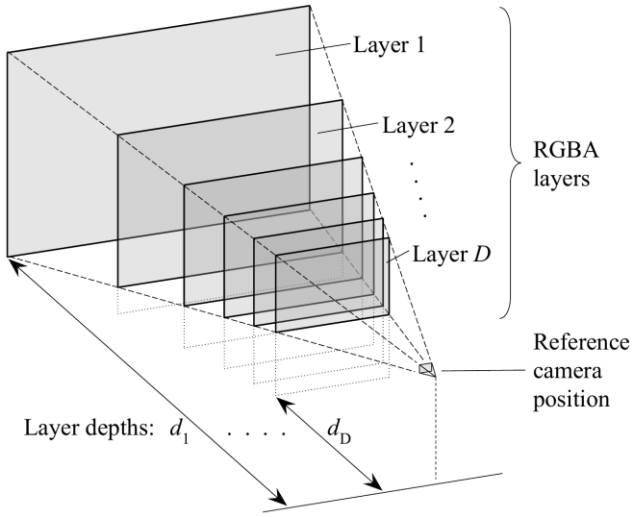
- Python 3.7.8
- Jax 0.2.19
- OpenCV 4.4.0

Note: To run the code on your own Google Pixel 4 data, please adjust the preprocessing step in `./code/util.py/load_data_and_calibration` if needed, such that the input dual pixel images are normalized to the range of [0, 1].

Multiplane Image(MPI) Representation

- MPI 의 핵심 아이디어는 특정 depth(defocus blur size), d 에 대한 fronto-parallel plane 에 대하여 depth(defocus blur size) 의 변화 $[d_1, \cdots, d_N]$ 따라 N 개의 fronto-parallel plane들을 집합인 MPI representation 으로 장면을 모델링 할 수 있다는 것임.
  - We model the scene using the MPI representation, previously used primarily for view synthesis. MPIs discretize the 3D space into N fronto-parallel planes at fixed depths (Fig. 5). We select depths corresponding to linearlychanging defocus blur sizes $[d_1, \cdots, d_N]$

즉, 하나의 영상내 특정 depth 을 가지는 픽셀들의 집합이 fronto-parallel layer 이며 이러한 fronto-parallel layer 들의 집합으로 넓은 범위의 depth 를 가지는 전체 영상을 표현 할 수 있어야 함.



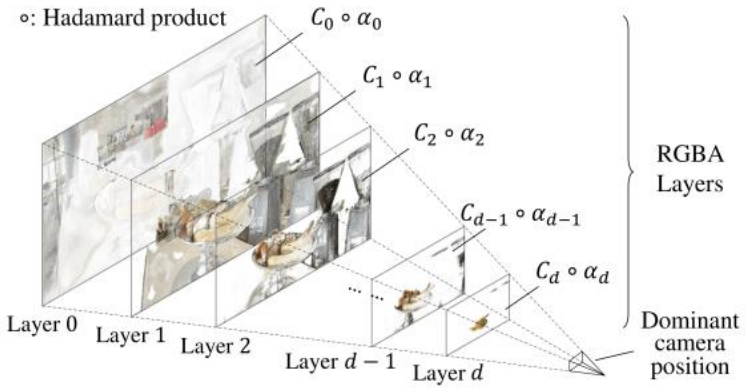


Figure 2: An illustration of MPI representation.

- 각 MPI plane 은 in-focus 영상 내 특정 depth 의 **intensity channel, c_i 과 alpha channel, α_i** (RGBA) 로 표현 할 수 있음.
  - Each MPI plane is an intensity-alpha image of the in-focus scene that consists of an intensity channel c_i and an alpha channel α_i.

Multiplane Image(MPI) Representation

All in focus image and Defocus image compositing

- 따라서, MPI plane 의 집합 $\{(C_1, \alpha_1), \cdots, (C_D, \alpha_D)\}$ 즉, MPI Representation 이 주어졌을 때 all-in-focus 영상, $\hat{I}_S$ 은 다음과 같이 표현 가능함.
  - Given an MPI, we composite the sharp image using the over operator: we sum all layers weighted by the transmittance of each layer, t_i.

$$\hat{I}_S = \sum_{i=1}^{N} t_i c_i = \sum_{i=1}^{N} [\alpha_i c_i \prod_{j=i}^{N} (1 - \alpha_j)]$$

i 번째 depth 에 해당하는 영역 (수정할 영역)

i 번째 depth 에 해당하지 않은 영역 (수정하지 않을 영역)

뿐만 아니라, 위 식에서 defocus map 도 pixel intensity 를 c_i 을 defocus blur size, d_i 로 대체하면 표현 가능함.

$$\hat{D} = \sum_{i=1}^{N} t_i d_i = \sum_{i=1}^{N} \alpha_i [d_i \prod_{j=i}^{N} (1 - \alpha_j)]$$

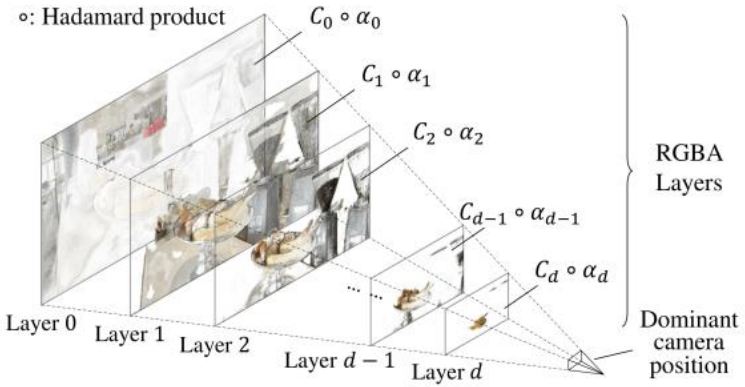


Figure 2: An illustration of MPI representation.

Multiplane Image(MPI) Representation

Defocus image rendering

- Defocus image, $\hat{I}_b^{\{l,r\}}$ 는 각 MPI plane 와 이에 대응되는 defocus kernel, $k_{d_i}^{\{l,r\}}$ 을 convolution 을 하여 defocus MPI layer 를 생성하고 이를 서로 다른 blur kernel 에 대한 MPI plane 들을 앞선 방식과 같이 결합하여 다음과 같이 표현 할 수 있음.
  - Given the left and right blur kernels $k_{d_i}^{\{l,r\}}$ for each layer, we render defocused images by convolving each layer with its corresponding kernel, then compositing the blurred layers.

$$\hat{I}_b^{\{l,r\}} = \sum_{i=1}^{N}[k_{d_i}^{\{l,r\}} * (c_i\alpha_i) \odot \prod_{j=i}^{N}(1 - k_{d_i}^{\{l,r\}} * \alpha_j)]$$

  - 실제로, 구현단계에서는 내가 원하는 특정 defocus kernel size, d_i 의 kernel 을 생성해 내기 위하여 defocus size d' 은 defocus size d 의 blur kernel 에 대하여 d'/d 만큼 scaling 을 통하여 구할 수 있다는 가정에 근거하여 defocus kernel size, d_i 가 까지 calibrated left and right kernel 을 scaling 함. 이어서, scaling 된 defocus kernel 을 MPI plane $c_i\alpha_i$ 에 convolution 하여 defocus MPI plane 을 얻어냄.
    - In practice, we scale the calibrated spatially-varying left and right kernels by the defocus size d_i, and apply the scaled spatially-varying blur to each intensity-alpha image $c_i\alpha_i$.

4.1. Multiplane Image (MPI) Representation

We model the scene using the MPI representation, previously used primarily for view synthesis [80, 91]. MPIs discretize the 3D space into N fronto-parallel planes at fixed depths (Fig. 5). We select depths corresponding to linearly-changing defocus blur sizes $[d_1, \ldots, d_N]$. Each MPI plane is an intensity-alpha image of the in-focus scene that consists of an intensity channel c_i and an alpha channel α_i.

All-in-focus image compositing. Given an MPI, we composite the sharp image using the *over* operator [53]: we sum all layers weighted by the transmittance of each layer t_i,

$$\hat{\mathbf{I}}_s = \sum_{i=1}^{N} t_i c_i = \sum_{i=1}^{N}\left[c_i\alpha_i \prod_{j=i+1}^{N}(1-\alpha_j)\right]. \quad (1)$$

Defocused image rendering. Given the left and right blur kernels $k_{d_i}^{\{l,r\}}$ for each layer, we render defocused images by convolving each layer with its corresponding kernel, then compositing the blurred layers as in Eq. (1):

$$\hat{\mathbf{I}}_b^{\{l,r\}} = \sum_{i=1}^{N}\left[\left(k_{d_i}^{\{l,r\}} * (c_i\alpha_i)\right) \odot \prod_{j=i+1}^{N}\left(1 - k_{d_j}^{\{l,r\}} * \alpha_j\right)\right], \quad (2)$$

where $*$ denotes convolution. In practice, we scale the calibrated spatially-varying left and right kernels by the defocus size d_i, and apply the scaled spatially-varying blur to each intensity-alpha image $c_i\alpha_i$. We note that we render left and right views from a single MPI, but with different kernels.

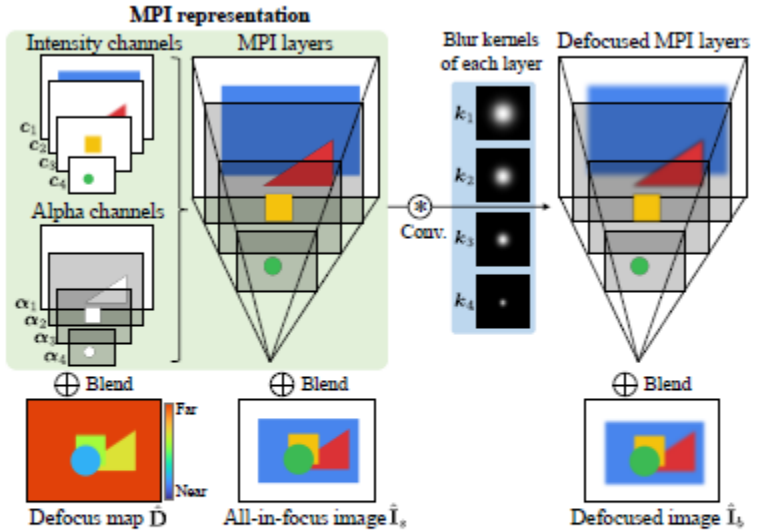


Figure 5: The multiplane image (MPI) representation consists of discrete fronto-parallel planes where each plane contains intensity data and an alpha channel. We use it to recover the defocus map, the all-in-focus image, and render a defocused image according to a given blur kernel.

Blur kernel scaling

```python
# Compute scaled blur kernels
scales = np.linspace(mpi_end_scale, mpi_start_scale, num=num_mpi_layers, endpoint=True)
optim_params['scales'] = scales
filter_halfwidth = blur_kernels.shape[-1] // 2
blur_kernels_scaled = util.rescale_blur_kernels(blur_kernels, filter_halfwidth*2+1, scales)

precomputed_vars = \
  dict(observations = observations,
       observations_volume = np.repeat(observations[None, ...], repeats=num_mpi_layers, axis=0),
       filter_halfwidth = filter_halfwidth,
       blur_kernels_scaled = blur_kernels_scaled,
       bias_correction = util.compute_bias_correction(observations, blur_kernels_scaled) )
```

```python
def rescale_blur_kernels(blur_kernels, blur_kernel_outsize, scales):
  """ Rescale blur kernels

  Args:
    blur_kernels: [..., F, F] calibrated filters

  Returns:
    blur_kernels_scaled: [#scales, ..., F, F] calibrated filters
  """
  blur_kernels_scaled = []
  for s in scales:
    filters_one_scale = rescale_blur_kernels_one_scale(blur_kernels, blur_kernel_outsize, 1 / s)
    blur_kernels_scaled.append(filters_one_scale)

  return np.stack(blur_kernels_scaled, axis=0)
```

```python
def rescale_blur_kernels_one_scale(blur_kernels, blur_kernel_outsize, scale):
  """Given filters of size (F, F), generates output filters of size (output_size, output_size),
  and scaled down by a factor of scale.
  Note that scale = 1 corresponds to the case when the filters are resized to output_size.

  Args:
    blur_kernels: [..., F, F]  multiple input filters of size (F, F), F should be odd
    blur_kernel_outsize: scalar, should be odd
    scale: scalar, when <0, the input is flipped about the center.

  Returns:
    [..., output_size, output_size] resampled filters of size (output_size, output_size)
  """

  if blur_kernel_outsize % 2 != 1:
    raise ValueError(f'output_size={blur_kernel_outsize} should be odd')
  F = blur_kernels.shape[-1]
  if F % 2 != 1:
    raise ValueError(f'Input dimensions should be odd but is {F}')

  flip = True if scale < 0 else False
  scale = abs(scale)

  # Translate these coordinates to the input coordinate space based on scale.
  base_scale = F / blur_kernel_outsize
  scale *= base_scale

  filter_halfsize = blur_kernel_outsize // 2
  output_pixel_centers = np.linspace(-filter_halfsize * scale, filter_halfsize * scale, blur_kernel_outsize)

  input_pixel_centers = np.linspace(-(F // 2), F // 2, F)

  if scale < 2.0:
    input_reshape = np.reshape(blur_kernels, (-1, F, F))

    def _unstack(x, axis=-1):
      """ unstack a numpy array along the input axis

      Args:
        x: a numpy array
        axis

      Returns:
        a list
      """
      return tuple(np.moveaxis(x, axis, 0))

    fs = [oscipy.interpolate.interp2d(input_pixel_centers, input_pixel_centers, inp, kind='linear', bounds_error=False, fill_value=0)
          for inp in _unstack(input_reshape, 0)]
    output = np.stack([f(output_pixel_centers, output_pixel_centers) for f in fs], axis=0)

    # Scale, so that resampled image sums to the same value.
    output = output * np.sum(input_reshape, (-1, -2), keepdims=True) / np.sum(output, (-1, -2), keepdims=True)
    output = np.reshape(output, blur_kernels.shape[:-2] + (blur_kernel_outsize, blur_kernel_outsize))

  else:
    x_out, y_out = np.meshgrid(output_pixel_centers, output_pixel_centers)
    x_out = np.reshape(x_out, (-1))
    y_out = np.reshape(y_out, (-1))

    # Sigma for Gaussian kernel used to filter the input.
    sigma = (scale - 1.0) / 2.0

    def _get_weights_1d(in_coords, out_coords):
      in_coords = np.tile(in_coords[..., None], (1, 1, blur_kernel_outsize * blur_kernel_outsize))
      in_coords = in_coords - out_coords[None, None, :]
      return np.exp(-(in_coords ** 2) / (2 * sigma * sigma))

    x_in, y_in = np.meshgrid(input_pixel_centers, input_pixel_centers)
    gx = _get_weights_1d(x_in, x_out)
    gy = _get_weights_1d(y_in, y_out)
    weights = gx * gy / (2.0 * np.pi * sigma * sigma)
    weights = np.moveaxis(weights, -1, 0)
    weights = np.reshape(weights, (blur_kernel_outsize * blur_kernel_outsize, F * F))

    input_reshape = np.moveaxis(blur_kernels, [-2, -1], [0, 1])  # [F, F, ...]
    input_vec = np.reshape(input_reshape, (F * F, -1))

    output_vec = np.matmul(weights, input_vec)

    # Scale, so that resampled image sums to the same value.
    output_vec = output_vec * np.sum(input_vec, 0) / np.sum(output_vec, 0)

    output = np.reshape(output_vec, (blur_kernel_outsize, blur_kernel_outsize) + (blur_kernels.shape[:-2]))
    output = np.moveaxis(output, [0, 1], [-2, -1])

  if flip:
    output = np.flip(np.flip(output, -1), -2)

  return output
```

Compose outputs from MPI

```
print(' ---> Start optimization ...')
outputs = optim.optimization([logit(mpi_init)], precomputed_vars, optim_params, patch_params)
```

```
def optimization(init, precomputed_vars, optim_params, patch_params):
  """MPI optimization

  Args:
    init: initialized MPI
    precomputed_vars: a dictionary of precomputed variables
    optim_params: a dictionary of optimization parameters
    patch_params: a dictionary of image patch parameters

  Returns:
    outputs: a dictionary containing the all-in-focus image and defocus map rendered from optimized MPI,
             as well as the MPI itself
  """

  # Generate outputs from optimized MPI
  loss_this, outputs, loss_terms_all = lossfun_helper(opt_get_params(opt_state), precomputed_vars, optim_params, patch_params)
  loss_data = loss_terms_all['loss_data']
  loss_aux_data = loss_terms_all['loss_aux_data']
  prior_sharp_im_tv = loss_terms_all['prior_sharp_im_tv']
  prior_alpha_tv = loss_terms_all['prior_alpha_tv']
  prior_entropy = loss_terms_all['prior_entropy']
  print(f'total loss: {loss_this:0.5e} | loss_data: {loss_data:0.5e} | loss_aux_data: {loss_aux_data:0.5e}')
  print(f'prior_sharp_im_tv: {prior_sharp_im_tv:0.5e} | prior_alpha_tv: {prior_alpha_tv:0.5e} | prior_entropy: {prior_entropy:0

  return outputs
```

```
def lossfun_helper(params, precomputed_vars, optim_params, patch_params):
  """Compute total loss of the current MPI

  Args:
    params: a dictionary of MPI
    precomputed_vars: a dictionary of precomputed variables
    optim_params: a dictionary of optimization parameters
    patch_params: a dictionary of image patch parameters

  Returns:
    loss: a scalar
    a dictionary of outputs (all-in-focus image & defocus map)
    a dictionary of each loss term
  """
```

```
intensity_scale_factor = 0.5 / np.mean(observations)

# ========== Compose outputs from MPI ==========
# sigmoid function makes sure both MPI colors and alphas are within the range [0, 1]
mpi = flax.nn.sigmoid(params[0])
mpi = np.pad(mpi, pad_width=((0,) * 2, (filter_halfwidth,) * 2, (filter_halfwidth,) * 2, (0,) * 2), mode='edge')
mpi_colors = mpi[..., :-1]
mpi_alphas = mpi[..., -1:]
sharp_im = multiplane_image.compose_sharp_image_from_mpi(mpi)
defocus_map = multiplane_image.compute_defocus_map_from_mpi(mpi, scales)
mpi_transmittance = multiplane_image.compute_layer_transmittance(mpi_alphas)
```

```
return loss, \
       {'sharp_im': sharp_im[filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth, :],
        'defocus_map': defocus_map[filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth],
        'mpi': mpi}, \
       {'loss_data': loss_data,
        'loss_aux_data': loss_aux_data,
        'prior_sharp_im_tv': prior_sharp_im_tv,
        'prior_alpha_tv': prior_alpha_tv,
        'prior_entropy': prior_entropy}
```

Compose output from MPI

```python
def compose_sharp_image_from_mpi(mpi):
  """ Compose all-in-focus images from MPIs.

  Args:
    mpi: [L, ..., H, W, C+1] num_layers, height, width, num_color_channels+1 (alpha channel)

  Returns:
    sharp_im: [..., H, W, C] sharp / all-in-focus images
  """
  mpi_colors = mpi[..., :-1]
  mpi_alphas = mpi[..., -1:]
  sharp_im = np.sum(compute_layer_transmittance(mpi_alphas) * mpi_colors, axis=0)

  return sharp_im
```

```python
def compute_defocus_map_from_mpi(mpi, defocus_scales):
  """ Compute defocus maps from MPIs.
  This is done in a similar way as composing sharp images from MPIs.

  Args:
    mpi: [L, H, W, C+1] num_layers, height, width, num_color_channels+1 (alpha channel)
    defocus_scales: a numpy array of blur kernel scales

  Returns:
    [H, W] defocus map
  """

  if mpi.shape[0] != defocus_scales.size:
    raise ValueError(f'MPI should have the same number of layers as kernel scales.')

  mpi_alphas = mpi[..., -1:]
  defocus_map = np.sum(compute_layer_transmittance(mpi_alphas) * defocus_scales[..., None, None, None], axis=0)

  return defocus_map[..., 0]
```

```python
def compute_layer_visibility(mpi_alphas):
  """Compute visibility of each MPI layer from alpha channels.
    The visibility of a pixel at i-th MPI layer is the product of (1-alpha) of all the layers in front of it, i.e:
      (1 - alpha_i+1) * (1 - alpha_i+2) * ... (1 - alpha_n-1)
    Args:
      mpi_alphas: [L, ..., H, W, 1] alpha channels for L layers, back to front.

    Returns:
      [L, ..., H, W, 1] layer visibility.
    """
  mpi_visibility = np.cumprod(1.0 - mpi_alphas[::-1, ...], axis=0)
  mpi_visibility = np.concatenate([np.ones_like(mpi_visibility[[0], ...]), mpi_visibility[:-1, ...]], axis=0)[::-1, ...]

  return mpi_visibility


def compute_layer_transmittance(mpi_alphas):
  """Compute transimittance of each MPI layer from alpha channels.
  The transmittance of a pixel at i-th MPI layer is the product of its own alpha value and (1-alpha) of all the layers in front of it, i.e:
      alpha_i * (1 - alpha_i+1) * (1 - alpha_i+2) * ... (1 - alpha_n-1)

  Args:
    mpi_alphas: [L, ..., H, W, 1] alpha channels for L layers, back to front.

  Returns:
    [L, ..., H, W, 1] The resulting transmittance.
  """

  return mpi_alphas * compute_layer_visibility(mpi_alphas)
```

Effect of Gaussian Noise and Defocus Estimation

- 앞에서 MPI representation 을 통하여 예측한 rendered defocus image, $\hat{I}_b^{\{l,r\}}$ 와 관측된 defocus image, $I_b^{\{l,r\}}$ 사이의 error, $\left\|\hat{I}_b^{\{l,r\}} - I_b^{\{l,r\}}\right\|_2^2$ 를 줄이는 방향으로 손실함수를 정의하여 최적화 할 수 있음.
  - Our goal is to optimize for an MPI with intensity-alpha layers$(\hat{C}_i, \hat{\alpha}_i)$, with defocus sizes di,$i \in [d_1, \cdots, d_N]$ such that the L2 loss $\left\|\hat{I}_b^{\{l,r\}} - I_b^{\{l,r\}}\right\|_2^2$ is minimized.
- 하지만, noise 의 영향으로 인하여 위와 같은 최적화 방식은 **더 작은 defocus size 에 편향(bias)되는 문제가 발생한다.** 따라서, 이를 해결하기 위하여, 이러한 bias 를 정량화 하고 bias correction term 을 통하여 최적화 과정에서 이러한 문제를 극복하도록 설계한다.
  - We show that, in the presence of image noise, minimizing the above loss biases the estimated defocus map towards smaller blur values. Specifically, we quantify this bias and then correct for it in our optimization.
- 편향(bias) 문제에 대한 **증명** 및 bias correction term **유도**
  - **가정**
    1. 관측된 blur 영상, $I_o^{\{l,r\}}$ 는 다음과 같이 noise, $N^{\{l,r\}}$ 와 noise-free image, $I_b^{\{l,r\}}$ 로 분해 될 수 있다.
    $$I_o^{\{l,r\}} = I_b^{\{l,r\}} + N^{\{l,r\}}$$
       - 이때, noise, $N^{\{l,r\}}$ 는 independent identically distributed 을 따르는 additive white noise, $\mathcal{N}(0, \sigma^2)$ 으로 가정 한다.
    2. 장면의 모든 content 는 정답 defocus size 가 d^* 인 하나의 단일 fronto-parallel plane 에 놓여져 있다고 가정한다.
       - For simplicity, we assume for now that all scene contents lie on a single fronto-parallel plane with ground truth defocus size d^*,
    3. 장면의 representation 은 defocus size 가 d_i 인 단일 opaque layer$(i.e., \hat{\alpha}_i = 1)$ (완전 불투명한 값을 가지는 layer 로 이해함)을 가지는 MPI layer 이다.

❖ independent and identically distributed, 독립 항등 분포)
- 두개 이상의 확률 변수를 함께 고려할 때, 이들의 확률적 특성이,
  1. 통계적 독립(independent) 이며,
  2. 동일한 확률분포(identically distributed) 를 가지고 있는 것을 말함.

Effect of Gaussian Noise on Defocus Estimation

- 편향(bias) 문제에 대한 **증명** 및 bias correction term **유도**
  - 확률에 음의 우도를 취하여 MPI 에 대한 MAP 를 유도하면 다음과 같음.

For simplicity, we assume for now that all scene contents lie on a single fronto-parallel plane with ground truth defocus size $d^\star$, and our scene representation is an MPI with a single opaque layer (i.e., $\hat{\boldsymbol{\alpha}}_i = 1$) with a defocus size hypothesis d_i. Under this assumption, the defocused image rendering equation (Eq. (2) of the main paper)

$$\hat{\mathbf{I}}_b^{\{l,r\}} = \sum_{i=1}^{N} \left[\left(\boldsymbol{k}_{d_i}^{\{l,r\}} * (\boldsymbol{c}_i \boldsymbol{\alpha}_i) \right) \odot \prod_{j=i+1}^{N} \left(1 - \boldsymbol{k}_{d_j}^{\{l,r\}} * \boldsymbol{\alpha}_j \right) \right] \tag{4}$$

reduces to

$$\hat{\mathbf{I}}_b^{\{l,r\}} = \boldsymbol{k}_{d_i}^{\{l,r\}} * \hat{\boldsymbol{c}}_i \,. \tag{5}$$

Similarly, Eq. (3) becomes:

$$\mathbf{I}_o^{\{l,r\}} = \boldsymbol{k}_{d^\star}^{\{l,r\}} * \boldsymbol{c}_i + \mathbf{N}^{\{l,r\}}. \tag{6}$$

We can express the above equations in the frequency domain as follows:

$$\boldsymbol{\mathcal{I}}_o^{\{l,r\}} = \boldsymbol{K}_{d^\star}^{\{l,r\}} \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^{\{l,r\}} \,, \tag{7}$$

where $\boldsymbol{\mathcal{I}}_o^{\{l,r\}}, \boldsymbol{K}_{d^\star}^{\{l,r\}}, \boldsymbol{C}_i$, and $\boldsymbol{\mathcal{N}}^{\{l,r\}}$ are the Fourier transforms of $\mathbf{I}_o^{\{l,r\}}, \boldsymbol{k}_{d^\star}^{\{l,r\}}, \boldsymbol{c}_i$, and $\mathbf{N}^{\{l,r\}}$, respectively. Note that the entries of $\boldsymbol{\mathcal{N}}^{\{l,r\}}$ are also independent identically distributed with the same Gaussian distribution $\mathcal{N}\left(0, \sigma^2\right)$ as the entries of $\mathbf{N}^{\{l,r\}}$.

We can obtain a maximum a posteriori (MAP) estimate of $\hat{\boldsymbol{C}}_i$ and d_i by solving the following optimization problem [12]:

$$\begin{aligned} &\arg\max P\left(\boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r | \hat{\boldsymbol{C}}_i, d_i, \sigma\right) P\left(\hat{\boldsymbol{C}}_i, d_i\right) \\ =&\arg\max P\left(\boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r | \hat{\boldsymbol{C}}_i, d_i, \sigma\right) P\left(\hat{\boldsymbol{C}}_i\right) \,. \end{aligned} \tag{8}$$

According to Eq. (7), we have

$$P\left(\boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r | \hat{\boldsymbol{C}}_i, d_i, \sigma\right) \propto \exp\left(-\frac{1}{2\sigma^2} \sum_{v=\{l,r\}} ||\boldsymbol{K}_{d_i}^v \hat{\boldsymbol{C}}_i - \boldsymbol{\mathcal{I}}_o^v||^2 \right) \,. \tag{9}$$

We also follow Zhou et al. [12] in assuming a prior for the latent all-in-focus image such that:

$$P\left(\hat{\boldsymbol{C}}_i\right) \propto \exp\left(-\frac{1}{2} ||\boldsymbol{\Phi} \hat{\boldsymbol{C}}_i||^2 \right) \,, \tag{10}$$

where we define $\boldsymbol{\Phi}$ such that

$$|\boldsymbol{\Phi}(f)|^2 = \frac{1}{\left|\hat{\boldsymbol{C}}_i(f)\right|^2} \,, \tag{11}$$

and f is the frequency. As $\hat{\boldsymbol{C}}_i$ is the unknown variable, we approximate Eq. (11) by averaging the power spectrum over a set of natural images $\{\boldsymbol{C}_i\}$:

$$|\boldsymbol{\Phi}(f)|^2 = \frac{1}{\int_{\boldsymbol{C}_i} |\boldsymbol{C}_i(f)|^2 \mu(\boldsymbol{C}_i)} \,, \tag{12}$$

where $\mu(\boldsymbol{C}_i)$ represents the probability distribution of $\boldsymbol{C}_i$ in image domain.

Maximizing the log-likelihood of Eq. (8) is equivalent to minimizing the following loss:

$$E\left(d_i | \boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r, \sigma\right) = \min_{\hat{\boldsymbol{C}}_i} \left(\sum_{v=\{l,r\}} ||\boldsymbol{K}_{d_i}^v \hat{\boldsymbol{C}}_i - \boldsymbol{\mathcal{I}}_o^v||^2 \right) + ||\sigma \boldsymbol{\Phi} \hat{\boldsymbol{C}}_i||^2 \,. \tag{13}$$

d_i can be estimated as the minimizer of the above energy function. Then given d_i, setting $\partial E / \partial \hat{\boldsymbol{C}}_i = 0$ yields the following solution of $\hat{\boldsymbol{C}}_i$, known as a *generalized Wiener deconvolution with two observations*:

$$\hat{\boldsymbol{C}}_i = \frac{\boldsymbol{\mathcal{I}}_o^l \overline{\boldsymbol{K}_{d_i}^l} + \boldsymbol{\mathcal{I}}_o^r \overline{\boldsymbol{K}_{d_i}^r}}{\left|\boldsymbol{K}_{d_i}^l\right|^2 + \left|\boldsymbol{K}_{d_i}^r\right|^2 + \sigma^2 |\boldsymbol{\Phi}|^2} \,, \tag{14}$$

where $\overline{\boldsymbol{K}_{d_i}^{\{l,r\}}}$ is the complex conjugate of $\boldsymbol{K}_{d_i}^{\{l,r\}}$, and $\left|\boldsymbol{K}_{d_i}^{\{l,r\}}\right|^2 = \boldsymbol{K}_{d_i}^{\{l,r\}} \overline{\boldsymbol{K}_{d_i}^{\{l,r\}}}$.

We then evaluate the defocus size hypothesis d_i by computing the minimization loss given the latent ground truth depth $d^\star$, and the noise level σ, that is,

$$E\left(d_i | \boldsymbol{K}_{d^\star}^l, \boldsymbol{K}_{d^\star}^r, \sigma\right) = \mathbb{E}_{\boldsymbol{C}_i, \boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r} E\left(d_i | \boldsymbol{K}_{d^\star}^l, \boldsymbol{K}_{d^\star}^r, \sigma, \boldsymbol{C}_i, \boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r\right) \tag{15}$$

$$= \mathbb{E}_{\boldsymbol{C}_i, \boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r} \left[\left(\sum_{v=\{l,r\}} ||\boldsymbol{K}_{d_i}^v \hat{\boldsymbol{C}}_i - \boldsymbol{\mathcal{I}}_o^v||^2 \right) + ||\sigma \boldsymbol{\Phi} \hat{\boldsymbol{C}}_i||^2 \right] \,, \tag{16}$$

where $\mathbb{E}(\cdot)$ is the expectation. Substituting $\hat{\boldsymbol{C}}_i$ with Eq. (14) gives us:

$$\begin{aligned} &E\left(d_i | \boldsymbol{K}_{d^\star}^l, \boldsymbol{K}_{d^\star}^r, \sigma\right) \\ =&\mathbb{E}_{\boldsymbol{C}_i, \boldsymbol{\mathcal{I}}_o^l, \boldsymbol{\mathcal{I}}_o^r} \left[\left(\sum_{v=\{l,r\}} ||\boldsymbol{K}_{d_i}^v \frac{\boldsymbol{\mathcal{I}}_o^l \overline{\boldsymbol{K}_{d_i}^l} + \boldsymbol{\mathcal{I}}_o^r \overline{\boldsymbol{K}_{d_i}^r}}{\left|\boldsymbol{K}_{d_i}^l\right|^2 + \left|\boldsymbol{K}_{d_i}^r\right|^2 + \sigma^2 |\boldsymbol{\Phi}|^2} - \boldsymbol{\mathcal{I}}_o^v||^2 \right) + ||\sigma \boldsymbol{\Phi} \frac{\boldsymbol{\mathcal{I}}_o^l \overline{\boldsymbol{K}_{d_i}^l} + \boldsymbol{\mathcal{I}}_o^r \overline{\boldsymbol{K}_{d_i}^r}}{\left|\boldsymbol{K}_{d_i}^l\right|^2 + \left|\boldsymbol{K}_{d_i}^r\right|^2 + \sigma^2 |\boldsymbol{\Phi}|^2}||^2 \right] \,. \end{aligned} \tag{17}$$

Then substituting $\boldsymbol{\mathcal{I}}_o^v$ with Eq. (7), we get:

$$\begin{aligned} &E\left(d_i | \boldsymbol{K}_{d^\star}^l, \boldsymbol{K}_{d^\star}^r, \sigma\right) \\ =&\mathbb{E}_{\boldsymbol{C}_i, \boldsymbol{\mathcal{N}}^l, \boldsymbol{\mathcal{N}}^r} \left[\left(\sum_{v=\{l,r\}} ||\boldsymbol{K}_{d_i}^v \frac{\left(\boldsymbol{K}_{d^\star}^l \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^l\right) \overline{\boldsymbol{K}_{d_i}^l} + \left(\boldsymbol{K}_{d^\star}^r \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^r\right) \overline{\boldsymbol{K}_{d_i}^r}}{\left|\boldsymbol{K}_{d_i}^l\right|^2 + \left|\boldsymbol{K}_{d_i}^r\right|^2 + \sigma^2 |\boldsymbol{\Phi}|^2} - \left(\boldsymbol{K}_{d^\star}^v \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^v\right)||^2 \right) + \right. \\ &\left. ||\sigma \boldsymbol{\Phi} \frac{\left(\boldsymbol{K}_{d^\star}^l \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^l\right) \overline{\boldsymbol{K}_{d_i}^l} + \left(\boldsymbol{K}_{d^\star}^r \boldsymbol{C}_i + \boldsymbol{\mathcal{N}}^r\right) \overline{\boldsymbol{K}_{d_i}^r}}{\left|\boldsymbol{K}_{d_i}^l\right|^2 + \left|\boldsymbol{K}_{d_i}^r\right|^2 + \sigma^2 |\boldsymbol{\Phi}|^2}||^2 \right] \,. \end{aligned} \tag{18}$$

Effect of Gaussian Noise on Defocus Estimation

- 편향(bias) 문제에 대한 **증명** 및 bias correction term **유도**
  - 확률에 음의 우도를 취하여 MPI 에 대한 MAP 를 유도하면 다음과 같음.

We now define $B = |K_{d_t}^l|^2 + |K_{d_t}^r|^2 + \sigma^2 |\Phi|^2$. We can rearrange the above equation as:

$$
\begin{aligned}
&E\left(d_i | K_{d^*}^l, K_{d^*}^r, \sigma\right) \\
=&\mathbb{E}_{C_t, \mathcal{N}^l, \mathcal{N}^r}\left[\left(\sum_{v=\{l,r\}} || \frac{C_i\left[K_{d_t}^v\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right) - K_{d^*}^v B\right]}{B} + \frac{K_{d_t}^v\left(\mathcal{N}^l \overline{K_{d_t}^l} + \mathcal{N}^r \overline{K_{d_t}^r}\right)}{B} - \mathcal{N}^v ||^2\right) + \right. \\
&\left. || \sigma\Phi \frac{C_i\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right)}{B} + \sigma\Phi \frac{\mathcal{N}^l \overline{K_{d_t}^l} + \mathcal{N}^r \overline{K_{d_t}^r}}{B} ||^2\right] . \qquad (19)
\end{aligned}
$$

Given that the entries of $\mathcal{N}^{\{l,r\}}$ are independent identically distributed with distribution $\mathcal{N}(0, \sigma^2)$, we have $\mathbb{E}(\mathcal{N}^v) = 0, \mathbb{E}\left(\mathcal{N}^{v2}\right) = \sigma^2$ and $\mathbb{E}\left(\mathcal{N}^l \mathcal{N}^r\right) = 0$, and we can simplify the above equation as:

$$
\begin{aligned}
&E\left(d_i | K_{d^*}^l, K_{d^*}^r, \sigma\right) \\
=&\mathbb{E}_{C_t, \mathcal{N}^l, \mathcal{N}^r}\left[\left(\sum_{v=\{l,r\}} || \frac{C_i\left[K_{d_t}^v\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right) - K_{d^*}^v B\right]}{B} ||^2 + || \frac{K_{d_t}^v\left(\mathcal{N}^l \overline{K_{d_t}^l} + \mathcal{N}^r \overline{K_{d_t}^r}\right)}{B} - \mathcal{N}^v ||^2\right) + \right. \\
&\left. || \sigma\Phi \frac{C_i\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right)}{B} ||^2 + || \sigma\Phi \frac{\mathcal{N}^l \overline{K_{d_t}^l} + \mathcal{N}^r \overline{K_{d_t}^r}}{B} ||^2\right] \qquad (20) \\
=&\mathbb{E}_{C_t}\left\{\left[\sum_{v=\{l,r\}} || \frac{C_i\left[K_{d_t}^v\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right) - K_{d^*}^v B\right]}{B} ||^2 + \sigma^2\left(|| \frac{K_{d_t}^{v\,2} + \sigma^2 |\Phi|^2}{B} ||^2 + || \frac{K_{d_t}^l K_{d_t}^r}{B} ||^2\right)\right] + \right. \\
&\left. || \sigma\Phi \frac{C_i\left(K_{d^*}^l \overline{K_{d_t}^l} + K_{d^*}^r \overline{K_{d_t}^r}\right)}{B} ||^2 + \sigma^2\left(|| \sigma\Phi \frac{K_{d_t}^l}{B} ||^2 + || \sigma\Phi \frac{K_{d_t}^r}{B} ||^2\right)\right\} . \qquad (21)
\end{aligned}
$$

Recall that, in Eq. (12), we defined $\Phi(f)$ such that $\frac{1}{|\Phi(f)|^2} = \int_{C_t} |C_i(f)|^2 \mu(C_i)$. Then we can further simplify $E\left(d_i | K_{d^*}^l, K_{d^*}^r, \sigma\right)$ as:

$$
\begin{aligned}
&E\left(d_i | K_{d^*}^l, K_{d^*}^r, \sigma\right) \\
=&\sum_f \left[\frac{\frac{1}{|\Phi|^2}\left|K_{d^*}^l K_{d_t}^r - K_{d^*}^r K_{d_t}^l\right|^2}{B}\right] + \sum_f \left[\frac{\frac{1}{|\Phi|^2}\sigma^2 |\Phi|^2\left(\left|K_{d^*}^l\right|^2 + \left|K_{d^*}^r\right|^2\right)}{B}\right] + \\
&\sum_f \left[\sigma^2\left(|| \frac{K_{d_t}^{l\,2} + \sigma^2 |\Phi|^2}{B} ||^2 + || \frac{K_{d_t}^{r\,2} + \sigma^2 |\Phi|^2}{B} ||^2 + 2|| \frac{K_{d_t}^l K_{d_t}^r}{B} ||^2 + || \sigma\Phi \frac{K_{d_t}^l}{B} ||^2 + || \sigma\Phi \frac{K_{d_t}^r}{B} ||^2\right)\right] \qquad (22) \\
=&\sum_f \left[\frac{\frac{1}{|\Phi|^2}\left|K_{d^*}^l K_{d_t}^r - K_{d^*}^r K_{d_t}^l\right|^2}{B}\right] + \sigma^2 \sum_f \left[\frac{\left|K_{d^*}^l\right|^2 + \left|K_{d^*}^r\right|^2}{B} + \frac{\sigma^2 |\Phi|^2}{B} + 1\right] \qquad (23) \\
=&\sum_f \left[\frac{\frac{1}{|\Phi|^2}\left|K_{d^*}^l K_{d_t}^r - K_{d^*}^r K_{d_t}^l\right|^2}{\left|K_{d_t}^l\right|^2 + \left|K_{d_t}^r\right|^2 + \sigma^2 |\Phi|^2}\right] + \sigma^2 \sum_f \left[\frac{\left|K_{d^*}^l\right|^2 + \left|K_{d^*}^r\right|^2 + \sigma^2 |\Phi|^2}{\left|K_{d_t}^l\right|^2 + \left|K_{d_t}^r\right|^2 + \sigma^2 |\Phi|^2} + 1\right] . \qquad (24)
\end{aligned}
$$

If we define $C_1\left(K_{d_t}^{\{l,r\}}, \sigma, \Phi\right) = \frac{\frac{1}{|\Phi|^2}}{\left|K_{d_t}^l\right|^2 + \left|K_{d_t}^r\right|^2 + \sigma^2 |\Phi|^2}$, and $C_2(\sigma) = \sigma^2 \sum_f 1$, then Eq. (24) boils down to Eq. (4) of the main paper.

$$
\begin{aligned}
&E\left(d_i \middle| K_{d^*}^{\{l,r\}}, \sigma\right) \\
&= \sum_f C_1\left(K_{d_i}^{\{l,r\}}, \sigma, \Phi\right)\left|K_{d^*}^l K_{d_i}^r - K_{d^*}^r K_{d_i}^l\right|^2 \\
&+ \sigma^2 \sum_f \left[\frac{\left|K_{d^*}^r\right|^2 + \left|K_{d_i}^l\right|^2 + \sigma^2 |\Phi|^2}{\left|K_{d^*}^l\right|^2 + \left|K_{d_i}^r\right|^2 + \sigma^2 |\Phi|^2}\right] + C_2(\sigma)
\end{aligned}
$$

The expected negative log-energy function corresponding to the MAP estimate of MPI

Effect of Gaussian Noise on Defocus Estimation

- 편향 문제에 대한 problem statement

Measure inconsistency between the hypothesized blur kernel, d_i and the true kernel, d^*

$$E\left(d_i \middle| K_{d^*}^{\{l,r\}}, \sigma\right) = \sum_f C_1\left(K_{d_i}^{\{l,r\}}, \sigma, \Phi\right) \left|K_{d^*}^l K_{d_i}^r - K_{d^*}^r K_{d_i}^l\right|^2 + \sigma^2 \sum_f \left[\frac{\left|K_{d^*}^r\right|^2 + \left|K_{d_i}^l\right|^2 + \sigma^2|\Phi|^2}{\left|K_{d^*}^l\right|^2 + \left|K_{d_i}^r\right|^2 + \sigma^2|\Phi|^2}\right] + C_2(\sigma)$$

Summation over all frequency

The expected negative log-energy function corresponding to the MAP estimate of MPI

$$K_{d^*}^{\{l,r\}}, K_{d_i}^{\{l,r\}} = Fourier\ tranfrom\ of\ kernel\ k_{d^*}^{\{l,r\}}, k_{d_i}^{\{l,r\}}, respetively.$$
$$\Phi = inverse\ spectiral\ power\ distribution\ of\ natural\ images.$$

First term: $d_i = d^*$ 일 때, 최소화

Second term: Depend noise variance and decrease as d_i decrease
Why? This is because, for a normalized blur kernel ($\left\|K_{d_i}^{\{l,r\}}\right\|_1 = 1$), as the defocus kernel size, $|d_i|$ decreases, its power spectrum, $\left\|K_{d_i}^{\{l,r\}}\right\|_2$ increase

따라서, 입력 영상에 white Gaussian noise 가 추가되면 더 작은 defocus size 에 편향(bias)되는 문제가 발생
This suggests that white Gaussian noise in input images results in a bias towards smaller blur kernels.

Effect of Gaussian Noise on Defocus Estimation

- 편향 문제를 해결하기 위한 bias correction term
  - 이러한 문제를 해결하기 위하여 다음과 같은 bias correction term 을 도입함.
    - Bias correction term 은 optimization loss 에서 이러한 편향을 막기 위하여 second term 의 근사값을 빼면서 목표를 달성 할 수 있음.
      - To account for this bias, we subtract an approximation of the second term, which we call the bias correction term, from the optimization loss.

$$\mathcal{B}\left(d_i \middle| K_{d^*}^{\{l,r\}}, \sigma\right) \approx \sigma^2 \sum_f \frac{\sigma^2 |\Phi|^2}{\left|K_{d_i}^l\right|^2 + \left|K_{d_i}^r\right|^2 + \sigma^2 |\Phi|^2}$$

- 이때, Second term 에 있는 d^* 관련된 term 을 bias correction 무시하는 이유는 d^* 가 그 자체로 작을 때에만 의미가 있기 때문에 무시 가능하다.
  - We ignore the terms containing ground truth d^*, as they are significant only when d^* is itself small.
- 더 나아가 defocus size 가 많은 즉, 좀 더 넓은 범위의 depth $[d_1, \cdots d_N]$ 에 대한 MPI 에 대해서는, 각 layer 마다 이에 해당하는 bias correction term, $\mathcal{B}(d_i)$ 을 뺌.

4.2. Effect of Gaussian Noise on Defocus Estimation

Using Eq. (2), we can optimize for the MPI that minimizes the L_2-error $||\hat{\mathbf{I}}_b^{\{l,r\}} - \mathbf{I}_o^{\{l,r\}}||_2^2$ between rendered images $\hat{\mathbf{I}}_b^{\{l,r\}}$ and observed DP images $\mathbf{I}_o^{\{l,r\}}$. Here we show that, in the presence of noise, this optimization is biased toward smaller defocus sizes, and we correct for this bias.

Assuming additive white Gaussian noise $\mathbf{N}^{\{l,r\}}$ distributed as $\mathcal{N}(0, \sigma^2)$, we can model DP images as:

$$\mathbf{I}_o^{\{l,r\}} = \mathbf{I}_b^{\{l,r\}} + \mathbf{N}^{\{l,r\}}, \tag{3}$$

where $\mathbf{I}_b^{\{l,r\}}$ are the latent noise-free images. For simplicity, we assume for now that all scene content lies on a single fronto-parallel plane with ground truth defocus size d^*. Then, using frequency domain analysis similar to Zhou *et al.* [88], we prove in the supplement that for a defocus size hypothesis d_i, the expected negative log-energy function corresponding to the MAP estimate of the MPI is:

$$E(d_i|K_{d^*}^{\{l,r\}}, \sigma) = \sum_f C_1(K_{d_i}^{\{l,r\}}, \sigma, \mathbf{\Phi}) \left| K_{d^*}^l K_{d_i}^r - K_{d^*}^r K_{d_i}^l \right|^2 + \sigma^2 \sum_f \left[\frac{|K_{d^*}^l|^2 + |K_{d^*}^r|^2 + \sigma^2 |\mathbf{\Phi}|^2}{|K_{d_i}^l|^2 + |K_{d_i}^r|^2 + \sigma^2 |\mathbf{\Phi}|^2} \right] + C_2(\sigma), \tag{4}$$

where $K_{d_i}^{\{l,r\}}$ and $K_{d^*}^{\{l,r\}}$ are the Fourier transforms of kernels $k_{d_i}^{\{l,r\}}$ and $k_{d^*}^{\{l,r\}}$ respectively, $\mathbf{\Phi}$ is the inverse spectral power distribution of natural images, and the summation is over all frequencies. We would expect the loss to be minimized when $d_i = d^*$. The first term measures the inconsistency between the hypothesized blur kernel d_i and the true kernel d^*, and is indeed minimized when $d_i = d^*$. However, the second term depends on the noise variance and decreases as $|d_i|$ decreases. This is because, for a normalized blur kernel ($||k_{d_i}^{\{l,r\}}||_1 = 1$), as the defocus kernel size $|d_i|$ decreases, its power spectrum $||K_{d_i}^{\{l,r\}}||_2$ increases. This suggests that white Gaussian noise in input images results in a bias towards smaller blur kernels. To account for this bias, we subtract an approximation of the second term, which we call the *bias correction term*, from the optimization loss:

$$\mathcal{B}\left(d_i|K_{d^*}^{\{l,r\}}, \sigma\right) \approx \sigma^2 \sum_f \frac{\sigma^2 |\mathbf{\Phi}|^2}{\left|K_{d_i}^l\right|^2 + \left|K_{d_i}^r\right|^2 + \sigma^2 |\mathbf{\Phi}|^2}. \tag{5}$$

We ignore the terms containing ground truth $d^\star$, as they are significant only when d^* is itself small, i.e., the bias favors the true kernels in that case. In an MPI with multiple layers associated with defocus sizes $[d_1, \ldots, d_N]$, we subtract per-layer constants $\mathcal{B}(d_i)$ computed using Eq. (5).

We note that we use a Gaussian noise model to make analysis tractable, but captured images have mixed Poisson-Gaussian noise [31]. In practice, we found it beneficial to additionally denoise the input images using burst denoising [32]. However, there is residual noise even after denoising, and we show in Sec. 5.1 that our bias correction term still improves performance. An interesting future research direction is using a more accurate noise model to derive a better bias estimate and remove the need for any denoising.

Bias correction term

- Bias correction term

```
precomputed_vars = \
  dict(observations = observations,
       observations_volume = np.repeat(observations[None, ...], repeats=num_mpi_layers, axis=0),
       filter_halfwidth = filter_halfwidth,
       blur_kernels_scaled = blur_kernels_scaled,
       bias_correction = util.compute_bias_correction(observations, blur_kernels_scaled) )
```

```
def compute_bias_correction(observations, blur_kernels_scaled):
  """ Computer bias correction term

  Args:
    observations: [#observations, H, W, C] height, width, num_color_channels
    blur_kernels_scaled: [L, #rows*#cols, #observations, F, F] scaled blur kernels, F: blur kernel size, assumed to be odd

  Returns:
    bias correction term [L, ]
  """

  def _my_ft2(x, axes=(-2, -1)):
    """2D Fourier Transform with fftshift

    Args:
      x: [H, W] spatial domain

    Returns:
      Resulting fourier transform
    """
    return np.fft.fftshift(np.fft.fft2(np.fft.ifftshift(x, axes=axes), axes=axes), axes=axes)

  blur_kernels_FT = _my_ft2(blur_kernels_scaled, axes=(-2, -1))
  C = 5e-3
  gaussian_noise_stv = 5e-3 ** 2 * 2
  K = np.sum(np.abs(blur_kernels_FT) ** 2, axis=-3)
  bias_correction = np.mean(observations) / 0.5 * gaussian_noise_stv * np.mean(C / (K + C), axis=(-3, -2, -1))

  return bias_correction
```

MPI optimization

- 목표
  - 최적의 MPI representation, $\{c_i, \alpha_i\}, i \in [1, \cdots, N]$ 을 예측이 목표. 최적의 MPI representation 을 예측하게 위하여 예측된 MPI representation 에 calibrated blur kernels 을 적용하여 defocus image 를 생성했을 때, 관측된 defocus image 와 최대한 같도록 하는 것이 목표임.
- 핵심 아이디어
  - Reconstruction loss 만으로 최적화 하기에는 **under-constrain 문제**가 발생한다. 따라서, 추가적인 prior loss 를 정의하여 constrain 을 더 강하게 걸어준다.
    - Under-constrain 이 발생하는 이유는 calibrated blur kernel 을 적용하여 **입력 영상과 같은 defocus image 를 생성 할 수 있는 MPI representation 의 경우의 수가 무수히 많기 때문**이다.
      - But minimizing only a reconstruction loss is insufficient: this task is ill-posed, as there exists an **infinite family of MPIs that all exactly reproduce the input images**.

⇒ 따라서, 1. bias-correction data term 2. Auxiliary data term 3. Intensity smoothness regularization term 4. Alpha and transmittance smoothness regularization term 5. Alpha and transmittance entropy regularization term 을 도입함.

$$\mathcal{L} = \mathcal{L}_{data} + \mathcal{L}_{aux} + \mathcal{L}_{intensity} + \mathcal{L}_{alpha} + \mathcal{L}_{entropy}$$

최종적으로 정의된 손실 함수

Bias-corrected data loss

- 기본 함수는 Charbonnier loss 를 사용 하되 이를 bias-corrected version 에 맞게 변형한 형태임.
  - Charbonnier loss, $l(x) = \sqrt{x^2/r^2 + 1}$ ⇒ Bias-correction version loss, $l(x, \mathcal{B}) = \sqrt{(x-\mathcal{B})^2/r^2 + 1}$

    we subtract an approximation of the second term, which we call the bias correction term
  - [목표] Bias-corrected data loss 의 목표는 noise 로 인하여 작은 defocus size 에 편향되는 문제를 해결하면서 동시에 궁극적인 목표인 예측된 영상과 관측된 입력 영상이 같도록 constrain 을 걸어 주는 것이다.
    - We use this loss function to form a data loss penalizing the difference between left and right input and rendered images.

$$\mathcal{L}_{data} = \sum_{(x,y)} l_{\mathcal{B}}\left(\hat{I}_b^{\{l,r\}}(x,y) - I_o^{\{l,r\}}(x,y), \mathcal{B}_{all}^{\{l,r\}}\right)$$

$$\mathcal{B}_{all}^{\{l,r\}} = \sum_{i=1}^{N}\left[k_{d_i}^{\{l,r\}} * \alpha_i \prod_{j=i}^{N}(1 - k_{d_i}^{\{l,r\}} * \alpha_j)\right]\mathcal{B}(d_i)$$

$\mathcal{B}_{all}^{\{l,r\}}$ 는 total bias correction term 으로 각 layer 마다 모든 bias correction term 의 합으로 이때, defocused transmittance, $t_i = \alpha_i \prod_{j=i}^{N}(1-\alpha_j)$ 가 가중치로 부여된 형태임.

We compute the total bias correction $\mathcal{B}_{all}^{\{l,r\}}$ all as the sum of all bias correction terms of each layer, weighted by the corresponding defocused transmittance.

Bias-corrected data loss. We consider the Charbonnier [11] loss function $\ell(x) = \sqrt{x^2/\gamma^2 + 1}$, and define a bias-corrected version as $\ell_{\mathcal{B}}(x, \mathcal{B}) = \sqrt{(x^2 - \mathcal{B})/\gamma^2 + 1}$, where we choose the scale parameter $\gamma = 0.1$ [6]. We use this loss function to form a data loss penalizing the difference between left and right input and rendered images as:

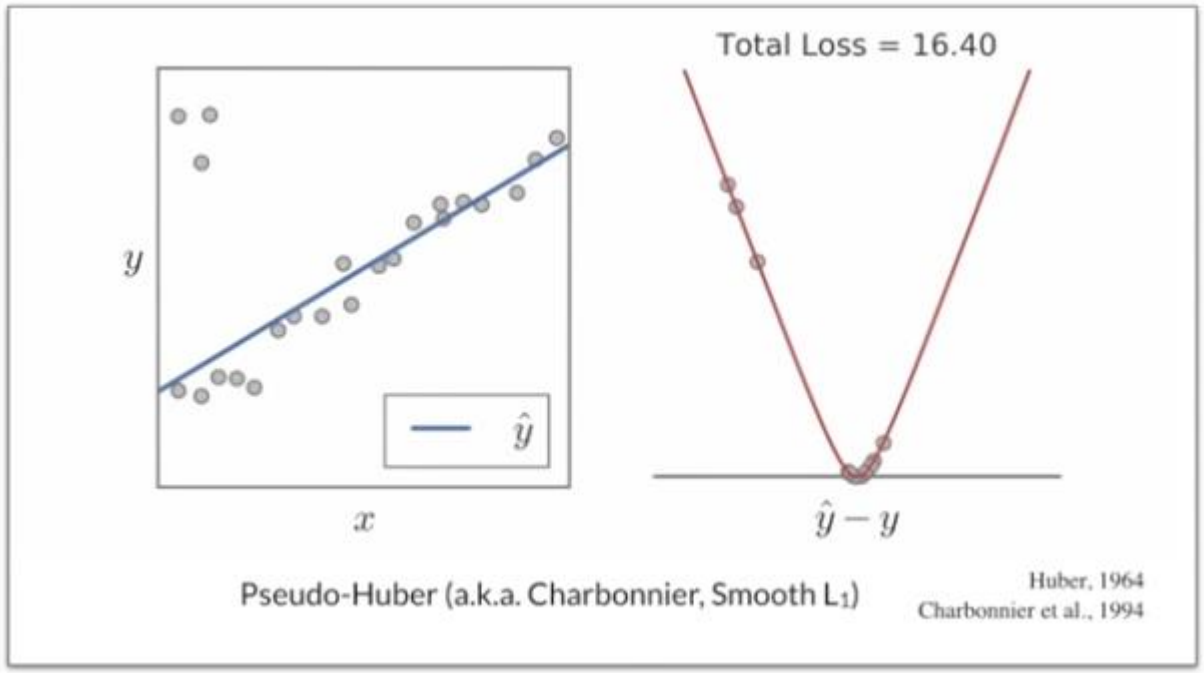
$$\mathcal{L}_{\text{data}} = \sum_{x,y} \ell_{\mathcal{B}}\left(\hat{\mathbf{I}}_b^{\{l,r\}}(x,y) - \mathbf{I}_o^{\{l,r\}}(x,y), \mathcal{B}_{\text{all}}^{\{l,r\}}\right), \quad (7)$$

$$\mathcal{B}_{\text{all}}^{\{l,r\}} = \sum_{i=1}^{N}\left[k_{d_i}^{\{l,r\}} * \alpha_i \prod_{j=i+1}^{N}\left(1 - k_{d_j}^{\{l,r\}} * \alpha_j\right)\right]\mathcal{B}(d_i). \quad (8)$$

We compute the total bias correction $\mathcal{B}_{\text{all}}^{\{l,r\}}$ as the sum of all bias correction terms of each layer, weighted by the corresponding defocused transmittance. Eq. (8) is equivalent to Eq. (2) where we replace each MPI layer's intensity channel $\boldsymbol{c}_i$ with a constant bias correction value $\mathcal{B}(d_i)$. To compute $\mathcal{B}(d_i)$ from Eq. (5), we empirically set the variance to $\sigma^2 = 5 \cdot 10^{-5}$, and use a constant inverse spectral power distribution $|\boldsymbol{\Phi}|^2 = 10^2$, following previous work [79].

Charbonnier loss

- 사용된 loss function
  a. Charbonnier loss(smoothness L1)



$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

- 동작 방식
  - L1 loss 와 L2 loss 의 혼합으로,
    - error 가 epsilon (일정 값) 보다 크면 L1 loss 로 작동.
      - steady gradients for large values of x
    - error 가 epsilon (일정 값) 보다 작으면 L2 loss 로 작동.
      - less oscillations during updates when x is small
- 의미
  - Error 가 충분히 작을 경우, 거의 맞는 것으로 취급하여 back-propagation 시에 loss 가 작게 전달됨. 즉, 정답에 가까이 있는 것에 대하여 중요도를 낮게 줌으로써 정답에 멀리 있는 sample 에 더 집중 할 수 있도록 함.
  - 따라서, 데이터 내에 noise 가 많은 경우 이를 학습 할 때 용이함.

```
def charbonnier_loss_from_L2_loss(x_square, gamma=1/10):
  """ isotropic total variatio on 2D data (e.g. single-channel images, or defocus maps)

  Args:
    x_square: L2 loss

  Returns:
    charbonnier loss
  """
  return np.sqrt(x_square / (gamma ** 2) + 1) - 1
```

https://forums.fast.ai/t/making-sense-of-charbonnier-loss/11978/5
https://ganghee-lee.tistory.com/33
https://stats.stackexchange.com/questions/351874/how-to-interpret-smooth-l1-loss

Bias-corrected data loss

```python
# ========== Bias-corrected data loss ==========
renderings_blurred, filtered_transmittance = multiplane_image.render_blurred_image_from_mpi(mpi, blur_kernels_scaled, patch_params)
cost_data_L2 = (renderings_blurred - observations) ** 2 - np.sum(filtered_transmittance * bias_correction[..., None, None, None, None], axis=0)
gamma = 1 / 10
loss_data = weight_loss_data * intensity_scale_factor * np.mean(np.mean(util.charbonnier_loss_from_L2_loss(cost_data_L2, gamma), axis=(0, -1)))
```

```python
def render_blurred_image_from_mpi(mpi, blur_kernels_scaled, patch_params):
  """ Render defocus-blurred images by first convolving each layer of an MPI with scaled spatially-varying blur kernels,
  and then blending all filtered layers.

  Args:
    mpi: [L, H, W, C+1] num_layers, height, width, num_image_channels+1 (alpha channel)
    blur_kernels_scaled: [L, #rows*#cols, #observations, F, F] resized filters, F: filter size
    patch_params: a dictionary of image patch parameters, useful for extracting image patches

  Returns:
    blurred_ims: [#observations, H, W, C] defocus-blurred image
    filtered_transmittance:  [L, H, W, C] filtered transmittance
  """

  mpi_colors = mpi[..., :-1]
  mpi_alphas = mpi[..., -1:]
  filtered_mpi = convolve_mpi_filter(np.concatenate([mpi_colors * mpi_alphas, mpi_alphas], axis=-1), blur_kernels_scaled, patch_params)

  filtered_colors = filtered_mpi[..., :-1]
  filtered_alphas = filtered_mpi[..., -1:]
  filtered_visibility = compute_layer_visibility(filtered_alphas)
  blurred_ims = np.sum(filtered_visibility * filtered_colors, axis=0)
  filtered_transmittance = filtered_visibility * filtered_alphas

  return blurred_ims, filtered_transmittance
```

```python
def convolve_mpi_filter(mpi, blur_kernels_scaled, patch_params):
  """ Convolve each layer of an MPI with scaled spatially-varying blur kernels.
  The amount of scale depends on the layer's defocus distance.

  Args:
    mpi: [L, H, W, C+1] num_layers, height, width, num_color_channels+1 (alpha channel)
    blur_kernels_scaled: [L, #rows*#cols, #observations, F, F] scaled blur kernels, F: blur kernel size, assumed to be odd
    patch_params: a dictionary of image patch parameters for extracting image patches

  Returns:
    filtered_mpi:  [L, #observations, H, W, C]
  """

  filter_halfwidth = blur_kernels_scaled.shape[-1] // 2
  num_observations = blur_kernels_scaled.shape[-3]

  mpi_patches = extract_patches(mpi, **patch_params, padding=filter_halfwidth)  # [#rows*#cols, L, H, W, C]
  mpi_patches = np.repeat(mpi_patches[..., None, :, :, :], repeats=num_observations, axis=-4)  # [#rows*#cols, L, #observations, H, W, C]

  filters_rescale_ = np.moveaxis(blur_kernels_scaled, 0, 1)
  filtered_patches = util.filter_image_batch(mpi_patches, filters_rescale_, 'valid')
  filtered_mpi = stitch_patches(filtered_patches, **patch_params, stitch_axis=(-3, -2))  # [L, #observations, H, W, C]

  return filtered_mpi
```

```python
def compute_layer_visibility(mpi_alphas):
  """Compute visibility of each MPI layer from alpha channels.
  The visibility of a pixel at i-th MPI layer is the product of (1-alpha) of all the layers in front of it, i.e:
    (1 - alpha_i+1) * (1 - alpha_i+2) * ... (1 - alpha_n-1)
  Args:
    mpi_alphas: [L, ..., H, W, 1] alpha channels for L layers, back to front.

  Returns:
    [L, ..., H, W, 1] layer visibility.
  """
  mpi_visibility = np.cumprod(1.0 - mpi_alphas[::-1, ...], axis=0)
  mpi_visibility = np.concatenate([np.ones_like(mpi_visibility[[0], ...]), mpi_visibility[:-1, ...]], axis=0)[::-1, ...]

  return mpi_visibility
```

Auxiliary data loss

- 문제점
  - 장면 내 하나의 pixel 은 하나의 layer 에만 속해야 한다. 하지만, 모델이 예측한 defocus image 은 $\hat{I}_b^{\{l,r\}} = \sum_{i=1}^{N}[k_{d_i}^{\{l,r\}} * (c_i\alpha_i) \odot \prod_{j=i}^{N}(1 - k_{d_i}^{\{l,r\}} * \alpha_j)]$ 으로 모든 layer 들의 sum 으로 정의되어 biased-corrected data loss, $\mathcal{L}_{data}$ 만으로는 하나의 scene content 가 하나의 layer 가 아닌 여러 개의 layer 들에 할당 될 수 있다.
    - In most real-world scenes, a pixel's scene content should be on a single layer. However, because the compositing operator of Eq. (2) forms a weighted sum of all layers, $\mathcal{L}_{data}$ can be small even when scene content is smeared across multiple layers.
- 해결 방법
  - 따라서, 이러한 layer 마다 제약 조건을 걸어주는 방식인 Auxiliary data loss 를 추가한다.

$$\mathcal{L}_{aux} = \sum_{(x,y),i} k_{d_i}^{\{l,r\}} * t_i(x,y) \odot l_{\mathcal{B}}(k_{d_i}^{\{l,r\}} * c_i(x,y) - I_o^{\{l,r\}}(x,y), \mathcal{B}(d_i))$$

we subtract an approximation of the second term, which we call the bias correction term

Auxiliary data loss 는 각 MPI layer 의 intensity 에 blurred transmittance 가중치가 곱해진 형태임.
To discourage this, we introduce a per-layer auxiliary data loss on each layer's intensity weighted by the layer's blurred transmittance

Auxiliary data loss. In most real-world scenes, a pixel's scene content should be on a single layer. However, because the compositing operator of Eq. (2) forms a weighted sum of all layers, $\mathcal{L}_{\text{data}}$ can be small even when scene content is smeared across multiple layers. To discourage this, we introduce a per-layer auxiliary data loss on each layer's intensity weighted by the layer's blurred transmittance:

$$\mathcal{L}_{\text{aux}} = \sum_{x,y,i} \left(k_{d_i}^{\{l,r\}} * t_i(x,y)\right) \odot \ell_{\mathcal{B}}\left(k_{d_i}^{\{l,r\}} * c_i(x,y) - \mathbf{I}_o^{\{l,r\}}(x,y), \mathcal{B}(d_i)\right), \quad (9)$$

where $\odot$ denotes element-wise multiplication. This auxiliary loss resembles the data synthesis loss of Eq. (7), except that it is applied to each MPI layer separately.

Auxiliary data loss

```python
# ========== Auxiliary data loss ==========
renderings_per_layer = multiplane_image.convolve_mpi_filter(mpi_colors, blur_kernels_scaled, patch_params)
cost_aux_data_L2 = (renderings_per_layer - observations_volume) ** 2 - bias_correction[..., None, None, None, None]
cost_aux_data_Charbonnier = lax.stop_gradient(filtered_transmittance) * util.charbonnier_loss_from_L2_loss(cost_aux_data_L2, gamma)
cost_aux_data_Charbonnier = scales.size * np.mean(cost_aux_data_Charbonnier, axis=(0, 1, -1))
loss_aux_data = weight_loss_aux_data * intensity_scale_factor * np.mean(cost_aux_data_Charbonnier)
```

```python
def convolve_mpi_filter(mpi, blur_kernels_scaled, patch_params):
  """ Convolve each layer of an MPI with scaled spatially-varying blur kernels.
  The amount of scale depends on the layer's defocus distance.

  Args:
    mpi: [L, H, W, C+1] num_layers, height, width, num_color_channels+1 (alpha channel)
    blur_kernels_scaled: [L, #rows*#cols, #observations, F, F] scaled blur kernels, F: blur kernel size, assumed to be odd
    patch_params: a dictionary of image patch parameters for extracting image patches

  Returns:
    filtered_mpi:  [L, #observations, H, W, C]
  """

  filter_halfwidth = blur_kernels_scaled.shape[-1] // 2
  num_observations = blur_kernels_scaled.shape[-3]

  mpi_patches = extract_patches(mpi, **patch_params, padding=filter_halfwidth)  # [#rows*#cols, L, H, W, C]
  mpi_patches = np.repeat(mpi_patches[..., None, :, :, :], repeats=num_observations, axis=-4)  # [#rows*#cols, L, #observations, H, W, C]

  filters_rescale_ = np.moveaxis(blur_kernels_scaled, 0, 1)
  filtered_patches = util.filter_image_batch(mpi_patches, filters_rescale_, 'valid')
  filtered_mpi = stitch_patches(filtered_patches, **patch_params, stitch_axis=(-3, -2))  # [L, #observations, H, W, C]

  return filtered_mpi
```

```python
def extract_patches(images, patch_size, num_rows, num_cols, padding=0):
  """ Divide images into image patches according to patch parameters

  Args:
    images: [..., #rows * P, #cols * P, C] height, width, #channels, P: patch size

  Returns:
    image_patches: [#rows * #cols, ..., P, P, C] The resulting image patches.
  """

  xv, yv = np.meshgrid(np.arange(num_cols), np.arange(num_rows))
  yv *= patch_size
  xv *= patch_size

  patch_size_padding = patch_size + 2 * padding
  xv_size, yv_size = np.meshgrid(np.arange(patch_size_padding), np.arange(patch_size_padding))

  yv_all = yv.reshape(-1)[..., None, None] + yv_size[None, ...]
  xv_all = xv.reshape(-1)[..., None, None] + xv_size[None, ...]
  patches = images[..., yv_all, xv_all, :]
  patches = np.moveaxis(patches, -4, 0)

  return patches
```

```python
def stitch_patches(patches, patch_size, num_rows, num_cols, stitch_axis):
  """ Stitch patches according to the given dimension

  Args:
    patches: [#rows * #cols, ..., P, P, C] / [#rows * #cols, ..., F, F]
    stitch_axis: (-3, -2) / (-2, -1)

  Returns:
    [..., #rows * P, #cols * P, C]  stitched images / [..., #rows * F, #cols * F] stitched kernels
  """

  axis_row, axis_col = stitch_axis
  patches_reshape = np.reshape(patches, (num_rows, num_cols, *patches.shape[1:]))
  patches_reshape = np.moveaxis(patches_reshape, (0, 1), (axis_row - 2, axis_col - 1))
  new_shape = onp.array(patches.shape[1:])
  new_shape[axis_row] *= num_rows
  new_shape[axis_col] *= num_cols
  images = np.reshape(patches_reshape, new_shape)

  return images
```

Intensity smoothness prior

- 목표
  - All-in-focus 영상과 MPI intensity channel 이 smooth 하도록 하는 regularization loss term 임.
- 방법
  - Total variance loss 를 사용하되 edge 영역까지 smooth 하지 않도록 한다.

$$V_E(I,E) = l\big(V(I)\big) + (1-E)\odot l(V(I))$$

    - E = edge map, V = total varaince function, I = image.
- 따라서, 최종 intensity smoothness regularization term 은 다음과 같음.

$$\mathcal{L}_{intensity} = \sum_{(x,y)} V_E\left(\widehat{I_s}, E\left(\widehat{I_s}\right)\right) + \sum_{(x,y)} V_E\left(t_i c_i, E(t_i c_i)\right)$$

Intensity smoothness. Our first regularization term encourages smoothness for the all-in-focus image and the MPI intensity channels. For an image $\mathbf{I}$ with corresponding edge map $\boldsymbol{E}$, we define an edge-aware smoothness based on total variation $\boldsymbol{V}(\cdot)$, similar to Tucker and Snavely [80]:

$$\boldsymbol{V_E}(\mathbf{I}, \boldsymbol{E}) = \ell\left(\boldsymbol{V}(\mathbf{I})\right) + (1-\boldsymbol{E}) \odot \ell\left(\boldsymbol{V}(\mathbf{I})\right), \quad (10)$$

where $\ell(\cdot)$ is the Charbonnier loss. We refer to the supplement for details on $\boldsymbol{E}$ and $\boldsymbol{V}(\cdot)$. Our smoothness prior on the all-in-focus image and MPI intensity channels is:

$$\mathcal{L}_{\text{intensity}} = \sum_{x,y} \boldsymbol{V_E}\left(\hat{\mathbf{I}}_s, \boldsymbol{E}\left(\hat{\mathbf{I}}_s\right)\right) + \sum_{x,y,i} \boldsymbol{V_E}\left(\boldsymbol{t_i c_i}, \boldsymbol{E}\left(\boldsymbol{t_i c_i}\right)\right). \quad (11)$$

Alpha and transmittance smoothness prior

- 목표
  - Intensity smoothness 와 마찬가지로 alpha channel 과 transmittance 가 smooth 해 지도록 하는 smoothness regularization loss term 임.
- 방법
  - Total variance loss 를 사용하되 edge 영역까지 smooth 하지 않도록 한다.
- 따라서, 최종 alpha and transmittance smoothness regularization loss term 은 다음과 같음.

$$\mathcal{L}_{alpha} = \sum_{(x,y),i} V_E\left(\sqrt{\alpha_i} E\left(\widehat{I_s}\right)\right) + \sum_{(x,y)} V_E\left(\sqrt{t_i} E\left(\widehat{I_s}\right)\right)$$

Alpha and transmittance smoothness. We use an additional smoothness regularizer on all alpha channels and transmittances (sharpened by computing their square root), by encouraging edge-aware smoothness according to the total variation of the all-in-focus image:

$$\mathcal{L}_{\text{alpha}} = \sum_{x,y,i} \left[\boldsymbol{V_E}\left(\sqrt{\boldsymbol{\alpha}_i}, \boldsymbol{E}\left(\hat{\mathbf{I}}_s\right)\right) + \boldsymbol{V_E}\left(\sqrt{\boldsymbol{t}_i}, \boldsymbol{E}\left(\hat{\mathbf{I}}_s\right)\right)\right]. \quad (12)$$

Smoothness regularization

3.3. Edge-aware Total Variation Function

We first define a pixel-wise total variation function of a single-layer image $\mathbf{I}$ that in used in both the intensity smoothness prior $\mathcal{L}_{\text{intensity}}$ and the alpha and transmittance smoothness prior $\mathcal{L}_{\text{alpha}}$:

$$\boldsymbol{V}(\mathbf{I}) = \sqrt{\mathbf{I}^2 * g - (\mathbf{I} * g)^2}, \tag{25}$$

where g is a two-dimensional Gaussian blur kernel:

$$g = \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}. \tag{26}$$

Each "pixel" in $\boldsymbol{V}(\mathbf{I})(x, y)$ is equivalent to, for the 3×3 window surrounding pixel (x, y) in $\mathbf{I}$, computing the sample standard deviation (weighted by a Gaussian kernel) of the pixel intensities in that window. This follows easily from two facts: 1) as g sums to 1 by construction, $\mathbf{I} * g$ produces an image whose pixel intensities can be viewed as expectations of their surrounding 3×3 input patch; and 2) the standard deviation $\sqrt{\mathbb{E}[(X - \mathbb{E}[X])^2]}$ can be written equivalently as $\sqrt{\mathbb{E}[X^2] - \mathbb{E}[X]^2}$.

As done in prior work [9], we would like to encourage edge-aware smoothness in addition to minimizing total variation, so a bilateral edge mask is computed using this total variation:

$$\boldsymbol{E}(\mathbf{I}) = 1 - \exp\left(-\frac{\mathbf{I}^2 * g - (\mathbf{I} * g)^2}{2\beta^2}\right). \tag{27}$$

In this equation, β is set to $1/32$ (assuming pixel intensities are in $[0, 1]$). A joint total variation function that takes into account both the original and the edge-aware total variation is then defined as:

$$\boldsymbol{V}_{\boldsymbol{E}}(\mathbf{I}, \boldsymbol{E}) = \ell(\boldsymbol{V}(\mathbf{I})) + (1 - \boldsymbol{E}) \odot \ell(\boldsymbol{V}(\mathbf{I})). \tag{28}$$

Intensity smoothness and alpha transmittance prior

```python
# ========== Intensity smoothness prior ==========
beta = 1 / 32
sharp_im_tv = tv(np.mean(intensity_scale_factor * sharp_im, axis=-1), gamma)[filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth]
edge_mask = util.edge_mask_from_image_tv(lax.stop_gradient(sharp_im_tv), gamma, beta)
sharp_im_tv_bilateral = sharp_im_tv * (1 - edge_mask)
sharp_im_tv_per_layer = np.mean( (scales.size * lax.stop_gradient(mpi_transmittance)[..., 0] * tv_batch(np.mean(intensity_scale_factor * mpi_colors, axis
                                 [..., filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth], axis=0)
edge_mask_per_layer = util.edge_mask_from_image_tv(lax.stop_gradient(sharp_im_tv_per_layer), gamma, beta)
sharp_im_tv_per_layer_bilateral = sharp_im_tv_per_layer * (1 - edge_mask_per_layer)
prior_sharp_im_tv = weight_prior_sharp_im_tv * \
                    (np.mean(sharp_im_tv_bilateral + sharp_im_tv_per_layer_bilateral) + np.mean(sharp_im_tv + sharp_im_tv_per_layer))
```

```python
# ========== Alpha and Transmittance smoothness prior ==========
alpha_tv = tv_batch(np.mean(np.sqrt(mpi_alphas), axis=-1), gamma)[..., filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth] + \
           tv_batch(np.mean(np.sqrt(mpi_transmittance), axis=-1), gamma)[..., filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth]
edge_mask_volume = np.repeat(edge_mask[None, ...], repeats=alpha_tv.shape[0], axis=0)
alpha_tv_bilateral = alpha_tv * (1 - edge_mask_volume)
prior_alpha_tv = weight_prior_alpha_tv * (np.mean(alpha_tv_bilateral) + np.mean(alpha_tv))
```

```python
def edge_mask_from_image_tv(im_tv, gamma, beta):
  """ edge mask from image total variation

  Args:
    im_tv: charbonnier image total variation
    gamma: gamma value used for converting L2 total variation to charbonnier total variation
    beta: parameter for edge mask

  Returns:
    edge mask
  """

  return 1 - np.exp(-((im_tv + 1) ** 2 - 1) * (gamma ** 2) / (2 * beta ** 2))
```

```python
"""MPI Optimization functions
"""

import jax.numpy as np
import flax
from jax import jit, lax
import jax.experimental.optimizers
import functools
import multiplane_image
from util import isotropic_total_variation as tv
from util import isotropic_total_variation_batch as tv_batch
import util
```

```python
def charbonnier_loss_from_L2_loss(x_square, gamma=1/10):
  """ isotropic total variatio on 2D data (e.g. single-channel images, or defocus maps)

  Args:
    x_square: L2 loss

  Returns:
    charbonnier loss
  """
  return np.sqrt(x_square / (gamma ** 2) + 1) - 1


def isotropic_total_variation(I, gamma):
  """ isotropic total variatio on 2D data (e.g. single-channel images, or defocus maps)

  Args:
    I: [H, W]

  Returns:
    [H, W] per-pixel total variation
  """

  gauss_1d = np.array([1, 2, 1])
  f_tv = gauss_1d[:, None] @ gauss_1d[None, :]
  f_tv = f_tv / np.sum(f_tv)

  I_blur = scipy.signal.convolve2d(I, f_tv, 'same')
  I_sq_blur = scipy.signal.convolve2d(I ** 2, f_tv, 'same')
  isotropic_tv = np.abs(I_sq_blur - I_blur ** 2)

  return charbonnier_loss_from_L2_loss(isotropic_tv, gamma)
```

Alpha and transmittance entropy prior

- 목표 및 방식
  - Scene content 가 여러 개의 layer 들이 아닌 하나의 layer 에만 집중되도록 하기 위하여 alpha 와 transmittance channel 에 가해주는 entropy 로 collision entropy 를 통하여 확률벡터, x 에 대하여 하나의 element 를 제외하고 나머지 모든 element 가 0 이 되도록 함.
    - The last regularizer is a collision entropy penalty on alpha channels and transmittances.
    - Minimizing collision entropy encourages sparsity: $S(x)$ is minimum when all but one elements of x are 0, which in our case encourages scene content to concentrate on a single MPI layer, rather than spread across multiple layers.
  - Collision entropy

$$S(x) = -log \, {\|x\|_2^2}/{\|x\|_1^2}$$

    - 하나의 element 를 제외하고 나머지 모든 element 가 0 이 되도록 하여 sparse 하게 만들어 주는 entropy 임.
    - 확률에 제곱을 하고 나서 negative log entropy 를 취하는 방식임.
  - 따라서, 최종적으로 정의된 entropy 는 다음과 같음.

$$\mathcal{L}_{entropy} = \sum_{(x,y)} S([\sqrt{\alpha_2}(x,y), \cdots, \sqrt{\alpha_N}(x,y)]^T) + \sum_{(x,y)} S([\sqrt{t_1}(x,y), \cdots, \sqrt{t_N}(x,y)]^T)$$

$\alpha_1 = 1$ 에 대해서는 생략한 이유
- 모든 scene content 가 가장 멀리 있는 layer 에서 종료된다고 가정하였기 때문에 가장 먼 MPI layer($\alpha_1 = 1$) 은 생략되었음.

  - $\alpha_i(x,y) = i$ 번째 layer 의 (x,y) pixel 에 대한 alpha channel.
  - $t_i(x,y) = i$ 번째 layer 의 (x,y) pixel 에 대한 transmittance channel.

Alpha and transmittance entropy. The last regularizer is a collision entropy penalty on alpha channels and transmittances. Collision entropy, defined for a vector x as $S(x) = -\log \|x\|_2^2 / \|x\|_1^2$, is a special case of Renyi entropy [68], and we empirically found it to be better than Shannon entropy for our problem. Minimizing collision entropy encourages sparsity: $S(x)$ is minimum when all but one elements of x are 0, which in our case encourages scene content to concentrate on a single MPI layer, rather than spread across multiple layers. Our entropy loss is:

$$\mathcal{L}_{\text{entropy}} = \sum_{x,y} S\left([\sqrt{\alpha_2}(x,y), \ldots, \sqrt{\alpha_N}(x,y)]^{\mathrm{T}}\right) + \sum_{x,y} S\left([\sqrt{t_1}(x,y), \ldots, \sqrt{t_N}(x,y)]^{\mathrm{T}}\right). \quad (13)$$

We extract the alpha channels and transmittances of each pixel (x, y) from all MPI layers, compute their square root for sharpening, compute a per-pixel entropy, and average these entropies across all pixels. When computing entropy on alpha channels, we skip the farthest MPI layer, because we assume that all scene content ends at the farthest layer, and thus force this layer to be opaque ($\alpha_1 = 1$).

Alpha and transmittance entropy prior

```
# ========== Entropy prior ==========
alpha_entropy = util.collision_entropy(np.sqrt(mpi_alphas[1:, ...]), axis=0)[filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth] + \
                util.collision_entropy(np.sqrt(mpi_transmittance[0:, ...]), axis=0)[filter_halfwidth:-filter_halfwidth, filter_halfwidth:-filter_halfwidth
prior_entropy = weight_prior_entropy * np.mean(alpha_entropy)

# total loss
loss = loss_data + loss_aux_data + prior_sharp_im_tv + prior_alpha_tv + prior_entropy
```

```
def collision_entropy(input, axis=0):
  """ Compute collision entropy of a tensor along given axis

  Args:
    input: a tensor
    axis

  Returns:
    resulting collision entropy
  """
  input /= np.sum(input, axis=axis)
  entropy = -np.log(np.sum(input ** 2, axis=axis))
  # numerical stability
  entropy = np.nan_to_num(entropy, nan=-onp.log(1 / input.shape[axis]))

  return entropy
```

Dataset

5. Experiments

We capture a new dataset, and use it to perform qualitative and quantitative comparisons with other state of the art defocus deblurring and defocus map estimation methods. The project website [85] includes an interactive HTML viewer [8] to facilitate comparisons across our full dataset.

Data collection. Even though DP sensors are common, to the best of our knowledge, only two camera manufacturers provide an API to read DP images—Google and Canon. However, Canon's proprietary software applies an unknown scene-dependent transform to DP data. Unlike supervised learning-based methods [1] that can learn to account for this transform, our loss function requires raw sensor data. Hence, we collect data using the Google Pixel 4 smartphone, which allows access to the raw DP data [16].

The Pixel 4 captures DP data only in the green channel. To compute ground truth, we capture a focus stack with 36 slices sampled uniformly in diopter space, where the closest focus distance corresponds to the distance we calibrate for, 13.7 cm, and the farthest to infinity. Following prior work [64], we use the commercial Helicon Focus software [35] to process the stacks and generate ground truth sharp images and defocus maps, and we manually correct holes in the generated defocus maps. Still, there are image regions that are difficult to manually inpaint, e.g., near occlusion boundaries or curved surfaces. We ignore such regions when computing quantitative metrics. We capture a total of 17 scenes, both indoors and outdoors. Similar to Garg *et al.* [24], we centrally crop the DP images to 1008×1344. We refer to the supplement for more details. Our dataset is available at the project website [85].

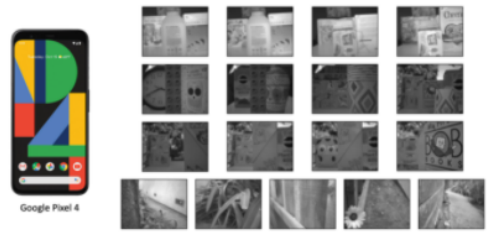
- 유일하게 raw data 를 제공하는 GooglePixel4 카메라로 촬영된 DP 영상을 사용함.
- Focal stack 을 활용해서 정답 영상 생성.
- 17개의 장면 수집.

Dataset

We captured a new dataset of 17 indoor and outdoor scenes using a Google Pixel 4 smartphone camera. Data can be found in `./DP_data_pixel_4`. Google Pixel 4 camera provides dual-pixel (DP) images in the green channel. These DP images are 14-bit, with a black level of 1024. Please refer to this GitHub Repo for more details about Google Pixel's DP data.

We also provide calibrated blur kernels and vignetting patterns of our device in `./DP_data_pixel_4/calibration`.

Dataset

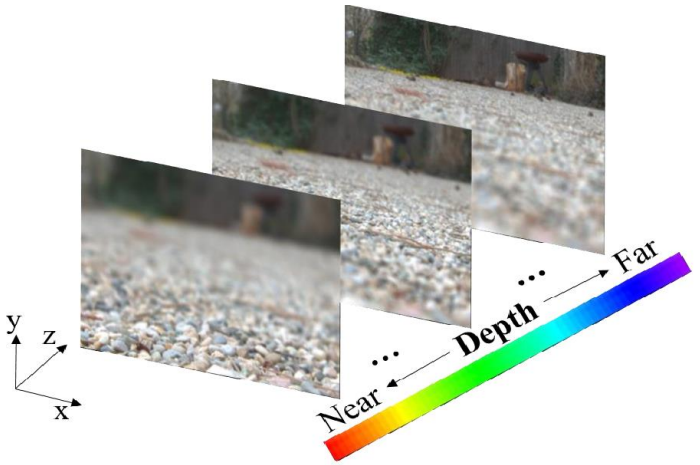


DP_data_pixel_4/01_left.png

DP_data_pixel_4/01_right.png

focal stack

- Focal stack is a set of images taken by the same camera but focused on different focal planes.
- Focal stacking 은 서로 다른 초점 거리에서 촬영한 여러 영상을 결합하여 결과 영상에 개별 소스 영상보다 더 큰 depth of field 를 제공하는 디지털 영상 처리 기술임.
- Focal stacking 은 개별 영상의 depth of field 가 매우 얕은 상황에서 사용 할 수 있음.
- 렌즈의 위치를 조정하여 focal planes 을 변화시키면 같은 영상에서 다른 depth of field 를 얻을 수 있음.

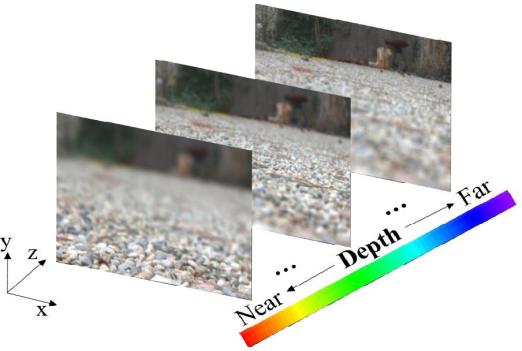


(a)Focal stack images (b)Post-capture refocusing

Fig. 1. Examples of focal stack images

focal stack

- 그렇다면, 어떻게 focal stack 으로 depth 를 찾을 수 있을까?
  - DFD(Depth From Defocus) 기법 활용.
  - Depth From Defocus is defined as the task of obtaining the depth of a scene from a focal stack.
  - DFD 는 focal stack 으로 부터 depth 를 찾는 방법임.
- DFD 기법의 한계점
  - Stack 이 만들어지는 과정 동안 장면에 변화가 없어야 함.
    - One limitation of using DFD is that the scene being imaged has to remain unchanged for the entire duration of the capture of the stack.
    - This precludes scenes or objects that are dynamic.



(a)Focal stack images

(b)Post-capture refocusing

Fig. 1. Examples of focal stack images

Result

5.1. Results

We evaluate our method on both defocus deblurring and depth-from-defocus tasks. We use $N = 12$ MPI layers for all scenes in our dataset. We manually determine the kernel sizes of the front and back layers, and evenly distribute layers in diopter space. Each optimization runs for 10,000 iterations with Adam [41], and takes 2 hours on an Nvidia Titan RTX GPU. We gradually decrease the global learning rate from 0.3 to 0.1 with exponential decay. Our JAX [9] implementation is available at the project website [85].

We compare to state-of-the-art methods for defocus deblurring (DPDNet [1], Wiener deconvolution [79, 88]) and defocus map estimation (DP stereo matching [82], supervised learning from DP views [24], DP defocus estimation based on kernel symmetry [64], Wiener deconvolution [79, 88], DMENet [45]). For methods that take a single image as input, we use the average of the left and right DP images. We also provide both the original and vignetting corrected DP images as inputs, and report the best result. We show quantitative results in Tab. 1 and qualitative results in Figs. 6 and 7. For the defocus map, we use the affine-invariant metrics from Garg *et al*. [24]. Our method achieves the best quantitative results on both tasks.

Defocus deblurring results. Despite the large amount of blur in the input DP images, our method produces deblurred results with high-frequency details that are close to the ground truth (Fig. 6). DPDNet makes large errors as it is trained on Canon data and does not generalize. We improve the accuracy of DPDNet by providing vignetting corrected images as input, but its accuracy is still lower than ours.

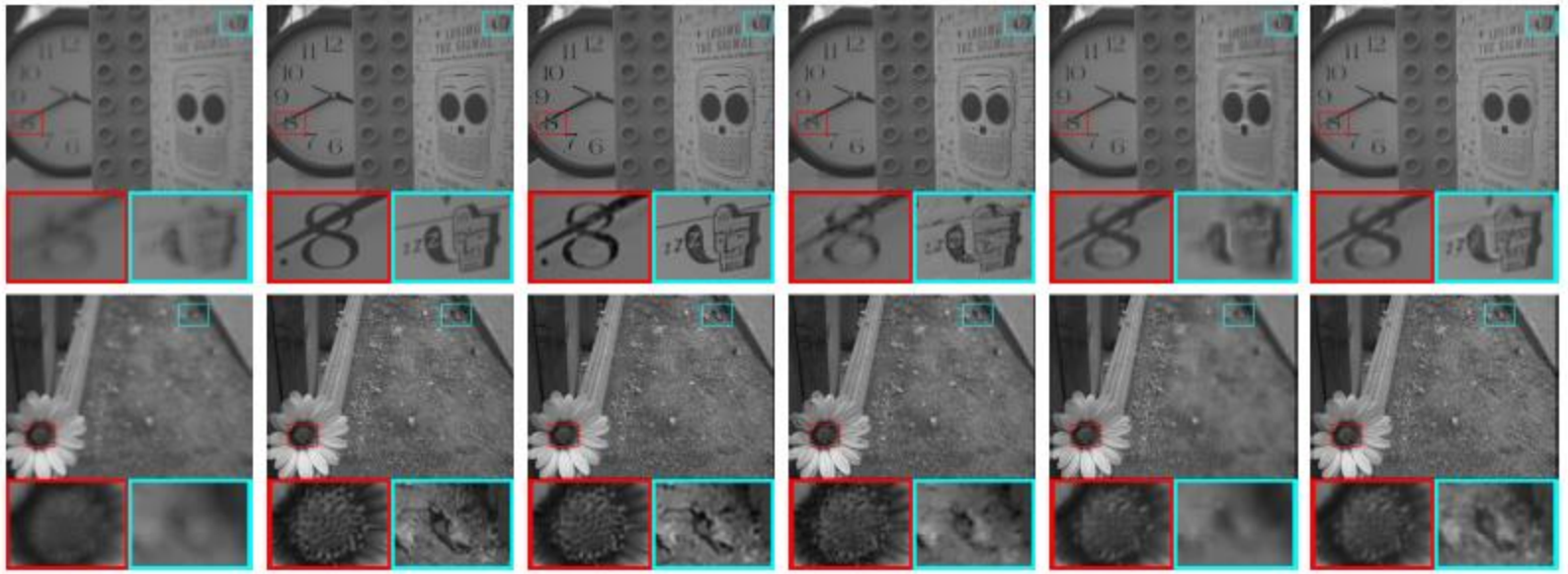
Defocus map estimation results. Our method produces defocus maps that are closest to the ground truth (Fig. 7), especially on textureless regions, such as the toy and clock in the first scene. Similar to [64], depth accuracy near edges can be improved by guided filtering [34] as shown in Fig. 7(d).

Ablation studies. We investigate the effect of each loss function term by removing them one at a time. Quantitative results are in Tab. 2, and qualitative comparisons in Fig. 8.

Our full pipeline has the best overall performance in recovering all-in-focus images and defocus maps. $\mathcal{L}_{\text{intensity}}$ and $\mathcal{L}_{\text{alpha}}$ strongly affect the smoothness of all-in-focus images and defocus maps, respectively. Without $\mathcal{L}_{\text{entropy}}$ or $\mathcal{L}_{\text{aux}}$, even though recovered all-in-focus images are reasonable, scene content is smeared across multiple MPI layers, leading to incorrect defocus maps. Finally, without the bias correction term $\mathcal{B}$, defocus maps are biased towards smaller blur radii, especially in textureless areas where noise is more apparent, e.g., the white clock area.

Results on Data from Abuolaim and Brown [1]. Even though Abuolaim and Brown [1] train their model on data from a Canon camera, they also capture Pixel 4 data for qualitative tests. We run our method on their Pixel 4 data, using the calibration from our device, and show that our recovered all-in-focus image has fewer artifacts (Fig. 9). This demonstrates that our method generalizes well across devices of the same model, even without re-calibration.

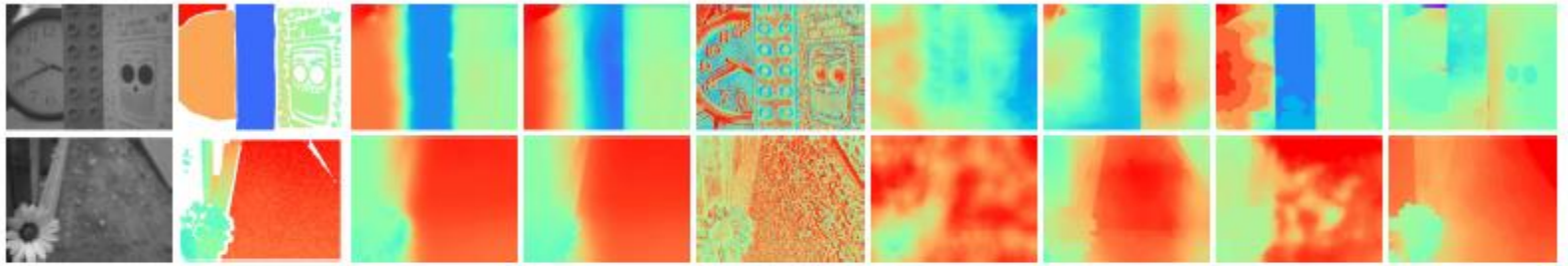
Result



(a) Input image (b) GT all-in-focus image (c) Ours (d) Wiener deconv. [88] (e) DPDNet [1] (Orig. Input) (f) DPDNet [1]

Figure 6: Qualitative comparisons of various defocus deblurring methods. Input images (a) shown as the average of two DP views, ground truth all-in-focus images (b) computed from focus stacks, recovered all-in-focus images (c) from our method and other methods (d)-(f). We improve the accuracy of DPDNet (e) trained on Canon data by providing vignetting corrected images (f). Our method performs the best in recovering high-frequency details and presents fewer artifacts.

Result



(a) Input image (b) Ground truth (c) Ours (d) Ours w/ GF (e) Wiener [88] (f) DMENet [45] (g) [64] (h) Garg [24] (i) Wadhwa [82]

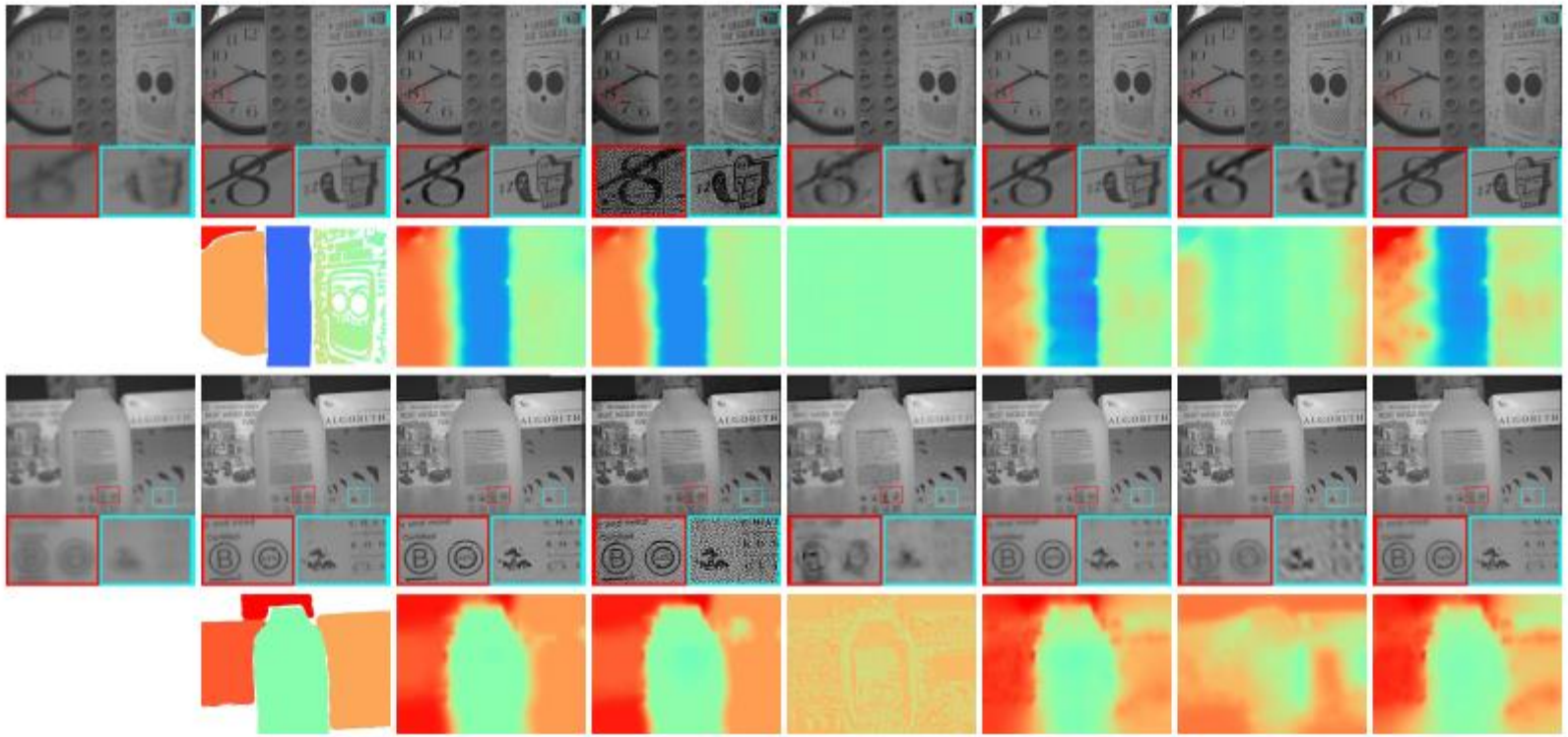
Figure 7: Qualitative comparisons of defocus map estimation methods. Input images (a) shown as the average of two DP views, ground truth defocus maps (b) from focus stacks with zero confidence pixels in white, our defocus maps (c), and our defocus maps with guided filtering (d), and defocus maps from other methods (f)-(i). Overall, our method produces results that are closest to the ground truth, and correctly handles textureless regions as well.

Result

Method	All-in-focus Image			Defocus Map		
	PSNR ↑	SSIM ↑	MAE ↓	AIWE(1) ↓	AIWE(2) ↓	$1-\lvert\rho_s\rvert$ ↓
Wiener Deconv. [88]	25.806	0.704	0.032	0.156	0.197	0.665
DPDNet [1]	25.591	0.777	0.034	-	-	-
DMENet [45]	-	-	-	0.144	0.183	0.586
Punnappurath *et al.* [64]	-	-	-	0.124	0.161	0.444
Garg *et al.* [24]	-	-	-	0.079	0.102	0.208
Wadhwa *et al.* [82]	-	-	-	0.141	0.177	0.540
Ours	26.692	0.804	0.027	0.047	0.076	0.178
Ours w/ guided filtering	26.692	0.804	0.027	0.059	0.083	0.193

Table 1: Quantitative evaluations of defocus deblurring and defocus map estimation methods on our DP dataset. "-" indicates not applicable. We use the affine-invariant metrics from [24] for defocus map evaluation. Our method achieves the best performance (highlighted in red) in both tasks.

Result



(a) Input image (b) Ground truth (c) Ours full (d) No $\mathcal{L}_{\text{intensity}}$ (e) No $\mathcal{L}_{\text{alpha}}$ (f) No $\mathcal{L}_{\text{entropy}}$ (g) No $\mathcal{L}_{\text{aux}}$ (h) No $\mathcal{B}$

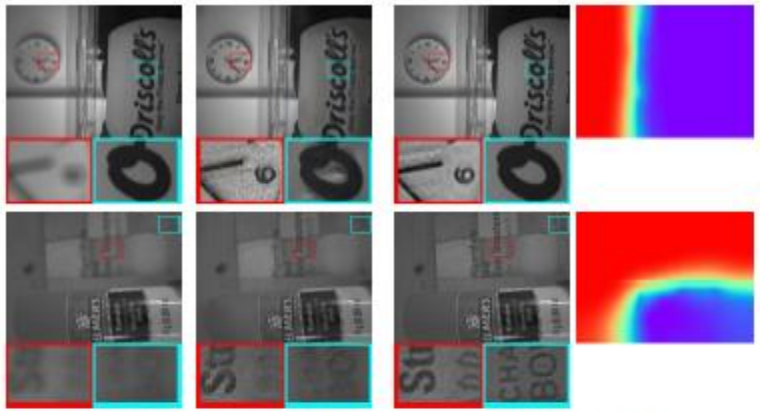
Figure 8: Ablation study. Input images (a), ground truth all-in-focus images, and defocus maps (b) with zero confidence pixels in white, our results (c), and our results with different terms removed one at a time (d)-(h). Removing $\mathcal{L}_{\text{intensity}}$ and $\mathcal{L}_{\text{alpha}}$ strongly affects the smoothness of all-in-focus images and defocus maps respectively. Results without entropy regularization $\mathcal{L}_{\text{entropy}}$, $\mathcal{L}_{\text{aux}}$, or the bias correction $\mathcal{B}$, exhibit more errors in defocus maps on textureless regions (clock).

Result

Method	All-in-focus Image			Defocus Map		
	PSNR ↑	SSIM ↑	MAE ↓	AIWE(1) ↓	AIWE(2) ↓	$1-\lvert\rho_s\rvert$ ↓
Full	26.692	0.804	0.027	0.047	0.076	0.178
No $\mathcal{L}_{\text{intensity}}$	14.882	0.158	0.136	0.047	0.078	0.185
No $\mathcal{L}_{\text{alpha}}$	24.748	0.726	0.037	0.161	0.206	0.795
No $\mathcal{L}_{\text{entropy}}$	27.154	0.819	0.026	0.057	0.085	0.190
No $\mathcal{L}_{\text{aux}}$	26.211	0.768	0.030	0.148	0.190	0.610
No $\mathcal{B}$	26.265	0.790	0.028	0.063	0.092	0.214

Table 2: Quantitative comparisons of ablation studies. We compare the full pipeline with removals of the regularization terms $\mathcal{L}_{\text{alpha}}$, $\mathcal{L}_{\text{intensity}}$ and $\mathcal{L}_{\text{entropy}}$, the auxiliary data loss $\mathcal{L}_{\text{aux}}$, and bias correction term $\mathcal{B}$ respectively. For all ablation experiments, we set the weights on remaining terms to be the same as the ones in the full pipeline. Best and second best results are highlighted in red and orange.

Result



(a) Input from [1] (b) DPDNet [1] (c) Our results

Figure 9: Results on data from [1]. Our method recovers all-in-focus images with fewer artifacts, while using the calibration data from our device.

Discussion

6. Discussion and Conclusion

We presented a method that optimizes an MPI scene representation to jointly recover a defocus map and all-in-focus image from a single dual-pixel capture. We showed that image noise introduces a bias in the optimization that, under suitable assumptions, can be quantified and corrected for. We also introduced additional priors to regularize the optimization, and showed their effectiveness via ablation studies. Our method improves upon past work on both defocus map estimation and blur removal, when evaluated on a new dataset we captured with a consumer smartphone camera.

Limitations and future directions. We discuss some limitations of our method, which suggest directions for future research. First, our method does not require a large dataset with ground truth to train on, but still relies on a one-time blur kernel calibration procedure. It would be interesting to explore blind deconvolution techniques [20, 48] that can simultaneously recover the all-in-focus image, defocus map, and unknown blur kernels, thus removing the need for kernel calibration. The development of parametric blur kernel models that can accurately reproduce the features we observed (i.e., spatial variation, lack of symmetry, lack of circularity) can facilitate this research direction. Second, the MPI representation discretizes the scene into a set of fronto-parallel depth layers. This can potentially result in discretization artifacts in scenes with continuous depth variation. In practice, we did not find this to be an issue, thanks to the use of the soft-blending operation to synthesize the all-in-focus image and defocus map. Nevertheless, it could be useful to replace the MPI representation with a continuous one, e.g., neural radiance fields [58], to help better model continuously-varying depth. Third, reconstructing an accurate all-in-focus image becomes more difficult as defocus blur increases (e.g., very distant scenes at non-infinity focus) and more high-frequency content is missing from the input image. This is a fundamental limitation shared among all deconvolution techniques. Using powerful data-driven priors to hallucinate the missing high frequency content (e.g., deep-learning-based deconvolution techniques) can help alleviate this limitation. Fourth, the high computational complexity of our technique makes it impractical for real-time operation, especially on resource-constrained devices such as smartphones. Therefore, it is worth exploring optimized implementations.