A Calibrated Defocus Simulator for Research in Passive Ranging

Citation

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37371763

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story
The Harvard community has made this article openly available. Please share how this access benefits you. Submit a story.

Accessibility
A Calibrated Defocus Simulator for Research in Passive Ranging

A DISSERTATION PRESENTED
BY
Dylan Zhou
TO
THE DEPARTMENT OF COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Bachelor of the Arts
IN THE SUBJECT OF
Computer Science and Physics

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2022
A Calibrated Defocus Simulator for Research in Passive Ranging

Abstract

Depth from defocus is a prominent area of computer vision research, yet unlike many other areas of the field, there is no dedicated dataset or general library for training and testing depth from defocus applications. As a result, researchers have had to manually capture data, which takes a tremendous amount of patience and precision, or they have resorted to approximations of real-world data using self-designed simulation pipelines—this simulated data is often simplistic or lacking in physical accuracy, which can adversely affect the performance of depth from defocus algorithms in deployment.

This thesis proposes a new pipeline for generating simulated data for depth from defocus, one that uses physics-based rendering and has controllable defocus that can be calibrated to the optical dimensions of any physical system. This new pipeline promises to deliver all the efficiency of synthetic data generation, to generate data that is faithful to real-world physics, and to be generalizable across different depth from defocus applications while also maintaining flexibility in its specificity for different scene and object types.

Here, I explore the power of physics-based rendering for generating realistically defocused images and I present experiments and demonstrations to showcase its functionalities, capabilities, and limitations when it comes to generating synthetic data for depth from defocus research.
## Contents

0 Introduction

  0.1 Motivation .................................................. 3
  0.2 Contributions ............................................... 6

1 Background

  1.1 An Overview of Depth Sensing ................................. 8
  1.2 Related Work in Depth From Defocus ......................... 9
  1.3 Physics-Based Rendering .................................... 16

2 A Forward Simulation Pipeline ................................. 20

  2.1 Metalens Renderer ............................................ 21
  2.2 Validation Renderer .......................................... 25
  2.3 Mitsuba Renderer ............................................ 28

3 Future Directions .............................................. 39

Appendix A Appendix ............................................. 41

  A.1 Converting Radial Depth to Conventional Depth ............ 41
  A.2 Proof: Rendered Image Depends Only on Ratio Between Sensor Width and Sensor Distance .................. 43

References ......................................................... 46
## Listing of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Depth maps with varying confidence thresholds.</td>
<td>3</td>
</tr>
<tr>
<td>1.1</td>
<td>Example of a depth map.</td>
<td>9</td>
</tr>
<tr>
<td>1.2</td>
<td>Computational tree.</td>
<td>14</td>
</tr>
<tr>
<td>1.3</td>
<td>Sampling over the full aperture to simulate depth of field.</td>
<td>18</td>
</tr>
<tr>
<td>1.4</td>
<td>Monte Carlo integration examples.</td>
<td>19</td>
</tr>
<tr>
<td>2.1</td>
<td>Point spread functions of the metalens.</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Metalens renderer rendering process.</td>
<td>23</td>
</tr>
<tr>
<td>2.3</td>
<td>How metalens renderer approximates occlusion events.</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Thin Lens Camera Model.</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Circle of Confusion.</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Thin Lens Model and Camera Parameters for Mitsuba.</td>
<td>31</td>
</tr>
<tr>
<td>2.7</td>
<td>Mitsuba Depth Map Before and After Conversion.</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>Initial comparisons between Python- and Mitsuba-generated defocus.</td>
<td>35</td>
</tr>
<tr>
<td>2.9</td>
<td>Difference between Mitsuba- and Python-rendered defocus for gradient.</td>
<td>36</td>
</tr>
<tr>
<td>2.10</td>
<td>Identical outputs for padded gradient.</td>
<td>37</td>
</tr>
<tr>
<td>2.11</td>
<td>Comparisons between Python- and Mitsuba-generated defocus on padded checker-board.</td>
<td>38</td>
</tr>
<tr>
<td>A.1</td>
<td>Converting Radial Depth to Conventional Depth</td>
<td>42</td>
</tr>
<tr>
<td>A.2</td>
<td>Ratio between sensor width and sensor distance determines rendered image</td>
<td>44</td>
</tr>
</tbody>
</table>
To my mom and dad, and to my brother Blair.
Acknowledgments

Special thanks to Professor Todd Zickler and Dr. Qi Guo (now Professor Qi Guo!) for advising my project, for showing me the magic of computer vision, for supporting me in my academic journey, for being patient through all my questions over the past two years, for meeting with me so often and on short notice, and for giving me such an exciting project to work on!

Thank you as well to Dean Hazineh for helping me debug my code on short notice when no one else was around.

Thank you to Professor Steven Gortler for reading my thesis!
Thank you to my friends for making my college experience so much fun.
Thank you to the Dunster House Community for being a second home for me.
And lastly, thank you to my family, to my mom and dad, and to my brother Blair, for always being there for me.
(I have much more to say, but I have to turn in this thesis very soon.)
Depth sensors have grown increasingly widespread in our daily lives, having become an essential component in modern smartphones, video game consoles, household robots, and even cars. Depth sensing, or ranging, is also a fundamental problem in computer vision and an active area of research in the field.

Passive solutions for depth sensing are particularly interesting because, like biological vision sys-
tems, they do not need to emit light to gain scene information. Within this passive sensing, depth from defocus—the topic of this thesis—holds tremendous promise due to its potential to be compact and computationally efficient.

Depth-from-defocus algorithms have existed for quite some time, with papers published as early as the 1980s\(^2\). Given two or more images of a scene—taken by a camera at a fixed position but with different focal lengths—depth from defocus exploits differences in the blurriness (or “defocus”) across images to estimate how close or far objects in the scene appear to be.

Recently, advances in nanofabrication techniques and in deep learning have enabled depth-from-defocus prototypes to come in smaller packages than ever while also retaining high computational efficiency. The state-of-the-art is the metalens depth sensor proposed by Guo et al.\(^9\) Inspired by the optics within the eyes of jumping spiders, the metalens depth sensor uses nanophotonic components to simultaneously capture two differently-defocused images and perform compact, efficient, and single-shot passive ranging. It has demonstrated depth recovery using a lens just 3 mm in diameter and an algorithm that computes depth maps at rates up to 100 fps while using fewer than 700 floating point operations per output pixel (FLOPs). For context, commercial alternatives require more than 7000 FLOPs for their algorithms.*

However, while these depth from defocus systems are highly compact and efficient, there is a tradeoff between the density of the predicted depth map and its accuracy. Given a scene, these systems output depth estimations for all parts of the scene but at varying confidence levels. But when we keep only the highest confidence regions, the remaining depth estimations are sparse, often covering less than 10% of the image (see Figure 1).

In my research, I am working towards a depth from defocus solution based on the state-of-the-art approach that will produce accurate depth estimations for 100% of the image while retaining as much speed and efficiency as possible. There are two parts to my solution: (1) a new synthetic data

\*An efficient implementation of a binocular stereo algorithm requires about 7000 FLOPs.\(^{10}\)
Measurements of several scenes with varying confidence thresholds, depth in meters. Depending on the image, once we impose a confidence threshold, the depth map can become quite sparse. In my research, my goal is to increase density while maintaining high accuracy and cheap deployment. (from [7])

This thesis focuses on the first part of that solution, the data generation pipeline. In this and the following chapters, I describe motivations behind this new approach, related work, technical details underlying my data generation pipeline, comparisons with other data generation methods, applications of my new data to existing depth from defocus algorithms, and implications and future work.

0.1 Motivation

Most depth from defocus algorithms today incorporate some amount of machine learning, which requires training data to calibrate parameters that enable the algorithm to make accurate predictions. The quality of an algorithm’s training data has a large effect on the quality of the predictions.
an algorithm will make, so a robust dataset is an essential ingredient for a good algorithm.

Currently, there are several issues with datasets being used to train depth from defocus algorithms. One problem is that there is no general library for depth from defocus, despite the many ongoing research projects in the area. In contrast, many other areas of computer vision research have large dedicated datasets for training: object and scene recognition, where the task is to predict a category label for an object or scene, has the ImageNet dataset; image segmentation, where the task is partitioning an image into distinct objects, has the Common Objects in Context (COCO) dataset; robotic manipulation, which deals with robots interacting with real-world objects, has the Yale-CMU-Berkeley (YCB) dataset. The closest thing to such libraries for depth from defocus is a dataset like the NYU Depth Dataset V2, which contains pairs of an all-in-focus image plus a depth map. But for depth from defocus, we need two or more differently defocused images of the same scene rather than a single all-in-focus image, so depth datasets like the NYU dataset do not suffice.

The absence of a dedicated dataset for depth from defocus can be attributed in part to the diversity of applications for depth sensing and the concept of “working ranges” for depth sensors. One research group might focus on depth sensing for self-driving cars, where the scale of objects is on the order of tens or hundreds of meters. Another group might specialize in miniature robotics, where the scale of objects is millimeters. These two groups would necessarily want distinct training datasets for fear of errors arising from dataset shift, which is caused by differences in the types of data an algorithm encounters in training versus in deployment. Moreover, it is crucial to maintain consistency between the camera hardware used to capture training data and the camera hardware used in deployment—not doing so would also lead to dataset shift—but between a ready-made dataset and novel research projects, it would be hard to use the same camera hardware for all the different depth from defocus applications. Furthermore, the data requirements for a given research project could be quite specific, for example, strict requirements on the image output size and resolu-
tion, or a precise set focus distances for each family of defocused images.

All these factors make it difficult to create a dedicated depth from defocus dataset, but the lack of one means that researchers have to generate large quantities of custom data consisting of scene images and depth maps. One approach is to capture data manually, that is, to set up scenes in the real world, take pictures, and measure depths. This approach is both costly and labor-intensive—since depth cannot be measured with a regular camera, researchers must acquire or build specialized equipment for sensing depth, which are typically comprised of multiple cameras or laser scanners, and both cameras and laser scanners are expensive. Furthermore, for the depths to correspond to the images, depth must be measured from the exact location of the camera, requiring high precision and care in the setup process. On top of that challenge, depth from defocus algorithms require, at the minimum, hundreds of images for training, making manually capturing data an even more tedious process.

Another approach, used by Guo et al.⁹, is to generate synthetic data. This approach escapes the costs and labor of manual data-gathering, but it introduces another problem: current synthetic data generation methods have trouble creating realistic scenes. For example, the training data in Guo et al.⁹ comprise of flat two-dimensional shapes layered on top of one another, and at occlusion boundaries between overlapping shapes the blur is an approximation that is close but not completely faithful to real-world optical effects. While this kind of data suffices for an application that makes quick but sparse depth predictions, training an algorithm meant to output a higher density depth map using such a simplistic dataset will limit not just its accuracy, but its generality and usability in the real world as well. And so, a second reason for the lack of a general dataset is the lack of realistic simulation methods.

Fortunately, the adjacent field of computer graphics, where the pursuit of realism has always been a primary motivation, offers a promising new path forward. Recent developments in physics-
based rendering methods—where algorithms simulate physical light transport, tracing paths of individual photons and modeling interactions between light and matter—have given rise to open-source software capable of generating simulated scene images that look more realistic than ever.

Mitsuba 2 \(^{17}\) (“Mitsuba”) is the latest such software. This thesis explores the power of Mitsuba in rendering realistically defocused images and presents experiments and demonstrations to showcase Mitsuba’s functionalities, capabilities, and limitations when it comes to generating synthetic data for depth from defocus research.

0.2 Contributions

The primary contribution of this thesis is to introduce a new data generation pipeline for depth from defocus research that uses physics-based rendering, produces physically accurate images, has controllable defocus that can be calibrated to the optical dimensions of any physical system. This new pipeline promises to deliver all the efficiency of synthetic data generation (as opposed to manual data capture) and to be generalizable across different depth from defocus applications but also flexible in its specificity for different scene and object types. That is, the same pipeline can generate data for different projects, but the types of objects and scenes in the data generated for each project can be customizable according to the different goals and working ranges of each project.
To best understand the present state of depth from defocus research and its requirements for data, it will be useful first to take a step back and survey depth sensing research at large, then to review related efforts in depth from defocus itself, and finally to examine physics-based rendering and its latest advancements.
1.1 An Overview of Depth Sensing

Depth sensing research holds a fundamental place in computer vision. As early as the 1980s, scientists and engineers began to study how to recover three-dimensional scene structure from two-dimensional images and built a variety of sensors and algorithms to produce reliable depth maps of their environments. Today, depth-sensing algorithms, like structure-from-motion, and depth-sensing hardware, like LiDAR sensors, are ubiquitous in modern technology—most smartphones come equipped with such features to enable high quality photography and smooth augmented reality experiences; many cars too have adopted them for improving safety and for powering self-driving capabilities.

The task of depth sensing can be formulated as follows: given a scene and a sensor, we want to extract a map that represents the distances from each point in the scene to the sensor. Figure 1.1 shows an example of a scene and its associated depth map. Broadly speaking, there are two main approaches to extracting this map and sensing depth: active and passive. Active depth sensing relies on emitting light to gain scene information, whereas passive sensing relies only on environmental light. Currently, the most popular form of active sensing is light detection and ranging, or LiDAR, where a source emits light—usually a grid of laser beams—and a sensor measures reflection times to determine scene depths. (Other forms of active sensing include time-of-flight (ToF) and structured light.) For passive sensing, the favored methods are depth from defocus—which uses the sharpness and blurriness of objects in the scene as cues for their depth—and learning-based monocular depth estimation—which trains on large datasets and uses deep neural networks to reconstruct depth from single images. Recently, with advances in deep learning, photonics, and nanofabrication technology, as well as demand for increasingly compact and efficient systems for low-powered and small devices, depth from defocus has proven to be a top candidate for a next generation of lightweight depth sensing.
1.2 Related Work in Depth From Defocus

Depth from defocus research has two main directions: learning-based approaches and physics-based approaches. Recently, there has been work to combine the two to develop hybrid solutions that have the accuracy of learning-based approaches with the speed and efficiency of physics-based approaches. (For my research and this thesis, we are more interested in the physics-based and hybrid solutions, although understanding learning-based approaches at a high level is still important.)

1.2.1 Learning-Based Approaches

Learning-based monocular depth estimation is its own area of research, but some methods have trickled over to depth from defocus. Most of these algorithms involve taking in a single image captured by a DSLR camera or equivalent and passing it into a deep neural network, which then outputs a depth map predicting the depths of each pixel in the image. One popular architecture
for these networks is the U-Net, where an encoder comprised of layers of convolutions, batch normalizations, and max pooling feeds into a decoder comprised of layers of convolutions, batch normalizations, and upsampling\textsuperscript{3,4}. While these networks typically perform extremely well and produce highly accurate depth maps, the tradeoff is that they are computationally expensive. As an example, Carvalho et al.\textsuperscript{3} use a Densenet-121 architecture\textsuperscript{12}, which has 8 million parameters and requires 3 billion FLOPs. Even more extreme, Chakrabarti et al.\textsuperscript{4} use the VGG-19 architecture\textsuperscript{20} as just one part of their network, and VGG-19 alone has 144 million parameters and requires 20 billion FLOPs. Compare these numbers to the 700 FLOPs needed for the metalens depth sensor to get a sense for how much more computation is being done here.

### 1.2.2 Physics-Based Approaches

While large neural networks offer a data-driven approach to recovering depth from defocus cues in images, another approach is to treat the depth recovery problem as an inverse image formation problem and to use physics principles to calculate depth for each pixel in an image. Here is how it works.\textsuperscript{*}

Consider a camera positioned at the origin of a coordinate system looking in the $+z$-direction. Suppose we add, at a distance $Z$ from the camera, a textured plane oriented in the $xy$-plane and centered on the $z$-axis. The orientation of this plane is called “fronto-parallel” to the camera. We can approximate the image $I(x, y)$ captured by this camera as a 2D convolution of a point spread function (also called a blur kernel) $k(x, y, Z)$ and a pinhole image $P(x, y)$:

$$I(x, y) = k(x, y, Z) * P(x, y), \quad (1.1)$$

where $(x, y)$ are pixel coordinates, $k(x, y, Z)$ is a function specific to the geometry of the camera and

\textsuperscript{*The following explanation and notation is based on Prof. Qi Guo’s Efficient Ranging with Computational Optics\textsuperscript{8}.}
its lens and dependent on the distance \( Z \), and the pinhole image \( P(x, y) \) refers to an ideal all-in-focus image of the scene.

Now suppose we capture two images \( I_1 \) and \( I_2 \) of the same scene but with different point spread functions \( k_1 \) and \( k_2 \) (this can be achieved by varying some camera parameters, such as changing the focus distance of the camera). Then by the convolution theorem, the ratio of the Fourier transform of \( I_1 \) and \( I_2 \) becomes a function of the depth \( Z \), independent of the pinhole image \( P \) and determined only by the optics of the camera setup:

\[
r_{\omega_x, \omega_y}(Z) = \frac{\mathcal{F}(I_1)(\omega_x, \omega_y)}{\mathcal{F}(I_2)(\omega_x, \omega_y)} = \frac{\mathcal{F}(k_1)(x, y, Z)}{\mathcal{F}(k_2)(x, y, Z)}.
\]  

We can almost always find a specific spatial frequency \( (\omega_x, \omega_y) \) for a given depth range such that \( r_{\omega_x, \omega_y}(Z) \) is a one-to-one mapping to depth \( Z \). Thus, by choosing \( (\omega_x, \omega_y) \) and pre-calibrating the function \( r_{\omega_x, \omega_y}(Z) \), we can measure depth from the defocused images with

\[
Z = r_{\omega_x, \omega_y}^{-1} \left( \frac{\mathcal{F}(I_1)(\omega_x, \omega_y)}{\mathcal{F}(I_2)(\omega_x, \omega_y)} \right).
\]  

In reality, most scenes are not fronto-parallel and so there is no global depth \( Z \). To account for this complication, many physics-based depth from defocus models assume images to have constant depths in patches and apply the same methods described above to estimate local depths within these patches. Instead of a global Fourier transform, this method approximates the same operation in local patches by convolving images with filters \( f(x, y) \) and \( g(x, y) \) that are localized in both space and frequency domain. Now, rather than a single \( Z \) for the entire scene, we have a dense depth map specified by

\[
Z(x, y) = \alpha \left( \frac{f * I_1(x, y)}{g * I_2(x, y)} \right),
\]  

where \( Z(x, y) \) is the depth estimated using a patch centered at \( (x, y) \), and the function \( \alpha \) is equiva-
lent to $r^{-1}$. Some examples of this formulation in practice are Watanabe and Nayer\textsuperscript{22}, Subbarao and Surya\textsuperscript{21}, and Farid and Simoncelli\textsuperscript{6}.

Most recently, Guo et al.\textsuperscript{9,10} present another variation of this formulation based on differential defocus. Assuming some camera with a Gaussian blur kernel, Guo et al. derive relations between the object depth $Z$ and knowable camera parameters:

\begin{align*}
    Z &= V/W, \quad (1.5) \\
    V &= (\Sigma^2 \mu_s^2 \dot{\rho}) \nabla^2 I, \quad (1.6) \\
    W &= (\Sigma^2 \mu_s \dot{\rho}) (\mu_s \rho - 1) \nabla^2 I - I, \quad (1.7)
\end{align*}

where $\Sigma$ is the width of the kernel at uniform magnification, $\mu_s$ is the separation between the sensor and lens, $\rho$ is the dioptric power of the lens (for a lens with focal length $f$, $\rho = 1/f$), $\dot{\rho}$ is the first time-derivative of the dioptric power $\rho$, and $\nabla^2 I$ and $I_t$ are spatial and temporal image derivatives, respectively.

In addition to a depth map, this approach also outputs a confidence map given by the formula:

\begin{equation}
    C = \left( \frac{\omega_0}{W^2} + \frac{V^2 \omega_1}{W^4} + \frac{V \omega_2}{W^5} + 1 \right)^{-1/2}, \quad (1.8)
\end{equation}

where $C \in [0, 1]$ and $\omega_0$, $\omega_1$, and $\omega_2$ are constants determined by the noise level, camera parameters, and derivative scale and order. Regardless of how good the depth predictor is, there will always be problematic locations where it fails (for example, it could be textureless regions, regions beyond the working range, points near depth discontinuities, etc.). The presence of a confidence map adds this important information about the quality of predictions for each pixel. The confidence map is a unique innovation that previous depth from defocus systems never had, and it can be used for tasks of varying complexity, from simply discarding low-confidence estimates, to improving depth
maps with edge-aware smoothing. For my project, I believe it will be critical for recovering a high-density high-accuracy depth map from the initial depth map predicted by equation (1.5).

Guo et al. implement their algorithm with a feed-forward computational tree with learned parameters that optimizes for the overall quality of the combined depth and confidence map (see Figure 1.2). In practice, Guo et al. build two different hardware solutions. The first is called Focal Track, which features a liquid membrane lens that is capable of deforming in response to electrical signals. Focal Track is a two-shot system that is capable of calculating depth maps at up to 100 fps and requiring less than 1000 FLOPs while also having a working range of over 75 cm. The second hardware solution is the metalens depth sensor, which uses nanophotonic components to simultaneously capture two differently-defocused images. The metalens depth sensor is a compact single-shot system that is less than 3 mm in diameter, has a working range of 10 cm, and calculates the depth map with fewer than 700 FLOPs.

Physics-based depth-from-defocus solutions strive for speed and efficiency. One direction research has taken is toward fabricating custom optics that are capable of modulating light in such a way that makes algorithmic computation simpler. Instead of producing conventional images, these optics produce “feature maps” of the environment—by virtue of their design, they essentially perform some of the computation, which was previously handled after image capture, at the speed of light the moment that light passes through them. These optics are called computational optics because they now share computation with the algorithm, and the overall paradigm is called computational visual sensing. Guo et al.’s deformable lens and metalens are examples of computational optics. By optimizing the design of these hardware components, we can develop faster and more efficient sensors.

Another direction for increasing speed and efficiency of defocus solutions is reducing the number of shots for acquiring the images $I_1$ and $I_2$ to one. The metalens depth sensor is possibly the
Figure 1.2: Computational tree. Lower nodes compute conventional Gaussian and Laplacian pyramids for each input pairs \((I_0, I_1)\), to obtain intermediate maps \((V_i, W_i)\) at each scale \(i\). Upper nodes apply a set of derivative filters \(\{\partial_{ij}^k\}\), and estimates per-\(ijk\) depth and confidence maps that are ultimately fused via softmax. Credit to Guo et al.\(^9\) for this figure.
first depth-from-defocus system to achieve this milestone—a feat made possible by state-of-the-art nanofabrication technology—and as a result, it is one of the fastest and most efficient depth-from-defocus sensors.

1.2.3 Hybrid Approaches

Hybrid approaches combine physics and learning. Qualitatively, hybrid solutions usually fall somewhere between physics-based and learning-based solutions when it comes to computational efficiency, as they often include computational optics to simplify their processing algorithms but they nonetheless have more complicated calculations than a physics-only model.

Wu et al. propose PhaseCam3D, a depth estimation system comprised of specialized front-end optics and a back-end reconstruction algorithm. The authors use the concept of the coded aperture and fabricate a custom phase mask which they insert directly in front of a camera lens. They use this setup to capture a coded image. This coded image then passes through a U-Net, which outputs an estimated depth for the scene. To achieve the best results, Wu et al. built an end-to-end architecture consisting of an optical layer with learnable parameters followed by the U-Net for depth prediction. This architecture allowed them to optimize not only the parameters of the U-Net but also to learn an optimal design for the phase mask—which they then fabricated and used in their prototype depth sensing system. The PhaseCam3D prototype demonstrates performance comparable to the Microsoft Kinect V2, one of the best ToF-based depth sensors on the mainstream market (Kinect produces smoother depth maps, but PhaseCam3D handles object boundaries better). At the same time, it requires no specialty light source and consumes less energy.

To train their system, Wu et al. used a dataset called FlyingThings3D, which is comprised of pairs of all-in-focus images accompanied by depth maps. Because they use custom PSFs for PhaseCam3D, Wu et al. digitally process the data and add custom blur to images from FlyingThings3D to simulate data captured by their own cameras. While a good approximation, this data could be
improved. First, a synthetic image where blur is added to a pinhole image will not be physically accurate, especially when there are occlusions and slanted planes in the scene. For example, in a real camera, blur at a specific point can have contributions from occluded scene points not visible in the pinhole image, which is not possible if blur is added retroactively. Second, the camera used to capture the original images will be different from the camera used in training and deployment, which will cause dataset shift. Although the phase mask is optimized given its current setup and current dataset, it is likely that it could be further optimized if the dataset were a set of images captured on the same camera used in training and deployment.

As seen in PhaseCam3D, Focal Track, and the metalens depth sensor, depth sensing systems that involve computational optics often feature uniquely designed cameras and lenses. Camera parameters can be quite specific to each system and yet they are a crucial part of each algorithm. Therefore, approaches based on computational optics need custom-tailored datasets for training and testing that accurately model a system's camera parameters while maintaining a high degree of realism. Here is where we turn to physics-based rendering and Mitsuba in particular.

1.3 Physics-Based Rendering

The goal of physics-based rendering is to generate photorealistic images of 3D scenes by using physics to model light-matter interactions as closely to the real world as possible. Physics-based rendering uses an algorithm interchangeably called “ray-tracing” or “path-tracing,” where rendering systems follow the path of a ray of light through a scene as it interacts with and bounces off objects in an environment. Ray tracers simulate cameras, ray-object intersections, light sources, visibility, surface scattering, indirect light transport, and ray propagation.

A basic ray tracer, which traces one ray per pixel, one shadow ray for every point light, and one
reflection or refraction ray per object intersection, can already generate fairly realistic images. How-
ever, these images typically have some discontinuities, which manifest as perfectly sharp silhouettes and shadow edges, perfectly clear mirror reflections, and perfect focus at all distances. In reality, images may have soft shadows, glossy reflections, and blur caused by depth of field or motion.

A key method for accurately simulating such effects is Monte Carlo integration, and modern physics-based renders like Mitsuba implement Monte Carlo ray-tracing algorithms to achieve photo-realism.

Take depth of field as an example. A basic ray tracer traces a single ray, which effectively simulates a pinhole camera and by definition generates an all-in-focus image. To simulate a real camera aperture with depth of field, the ray tracer should really compute an average (or integral) over all the light rays that pass through the aperture (see Figure 1.3). There are an infinite number of such light rays, but the integral in question can be closely approximated by summing a finite number of randomly sampled light rays. As the number of samples increases, the approximated integral approaches its true value. This is the general idea of Monte Carlo integration and Monte Carlo ray-tracing. For depth of field we sample over the aperture—similarly, for motion blur we can sample random times over the shutter interval (Figure 1.4a) and for area light we can sample over the light source (Figure 1.4b).
Figure 1.3: (Left) A basic ray tracer traces a single ray and reports the value of pixel \( I(x) = L(p, d(x)) \) as the radiance through point \( p \) along vector \( d(x) \). (Right) A more realistic ray tracer averages the contributions of all light rays that pass through the aperture \( D \).

\[
I(x) = L(p, d(x))
\]

\[
I(x) = \frac{1}{|D|} \int_D L(p, d(x, p)) dA(p)
\]
For more accurate motion blur, integrate over the shutter interval.

\[ I(x) = L(p, d(x), t_0) \]  
\[ I(x) = \frac{1}{|t_1 - t_0|} \int_{t_0}^{t_1} L(p, d(x), t) dt \]

**Fig. 1.4** (Top) Basic ray tracer vs. Monte Carlo ray tracer for motion blur. (Bottom) Basic ray tracer vs. Monte Carlo ray tracer for area lighting.

For more accurate lighting, integrate over the light source area.

\[ L(x, v) = \frac{I \cos \theta}{r^2} \]  
\[ L(x, v) = \frac{1}{|S|} \int_{S} \frac{I \cos \theta}{r^2} dA \]

(a) For more accurate motion blur, integrate over the shutter interval.

(b) For more accurate lighting, integrate over the light source area.

**Figure 1.4:** (Top) Basic ray tracer vs. Monte Carlo ray tracer for motion blur. (Bottom) Basic ray tracer vs. Monte Carlo ray tracer for area lighting.
In this chapter, I describe three different forward simulation pipelines. First is the metalen renderer, used by Guo et al. to generate flat scenes to train Focal Track and the metalens depth sensor. Second is a simple convolution-based renderer for fronto-parallel scenes that I used to validate my Mitsuba-generated images. Let’s call this the validation renderer. Third is the Mitsuba renderer.
2.1 Metalens Renderer

The metalens renderer generates defocused image pairs \((I_+, I_-)\) from a digital description of a virtual three-dimensional scene consisting of (1) a collection of planar shapes and associated texture patterns, (2) their positions on the image plane, and (3) a depth value for each pixel on the image plane. To model the lens, the renderer also takes as input a user-specified collection of point spread functions (PSFs) at a discrete set of spatial locations \(\{(X_k, Z_k)\}\). We denote the PSF at location \((X, Z)\) as \(k(u; X, Z)\), where \(u\) is the pixel location on the photosensor of the origin of the PSF. For a metalens, the nanophotonics at the lens surface can make it such that there are different PSFs at different points on the lens depending on where the light rays pass through, whereas for a conventional lens, the PSFs may all be disk-shaped blur kernels (also called “pillbox” blur kernels in computer vision literature). (See Figure 2.1 for an example of PSFs measured by the metalens sensor.)

Next, for simplicity, we describe the rendering process for a 2-dimensional slanted line segment—the process generalizes to more complicated shapes in 3 dimensions, as they can be decomposed into piecewise linear segments. Figure 2.2 illustrates this rendering process.

As a first step, the metalens renderer uses bilinear interpolation between the locations at which the PSF was specified to approximate the PSF at each vertex of the line segment. We write the two vertices as \(\{(X^v, Z^v)\}_{v=0,1}\) and the associated PSF as \(k(u; X^v, Z^v)\). Then, the renderer approximates the PSF \(k(u; X, Z)\) at every point \((X, Z)\) along the line segment using linear interpolation based on the PSFs at each vertex \((X^v, Z^v)\) with a weighting function \(w_v\left(\frac{x}{Z}\right)\). The final image \(I(x)\) is computed by summing all the PSFs at all points on the line segment, multiplied by the spatial texture (emitted radiance) pattern \(P\) that exists on the segment:

\[
I(x) \approx \sum_{X,Z} \sum_v k\left(x - \frac{X}{Z}; X^v, Z^v\right) w_v\left(\frac{X}{Z}\right) P\left(\frac{X}{Z}\right).
\] (2.1)
Figure 2.1: Point spread functions of the metasensor, measured using LED and laser sources. In general, the shapes of the PSFs resemble pillboxes, but with ringing. The PSFs from the LED sources are subject to chromatic aberration and are therefore asymmetric.
Figure 2.2: Rendering defocused images of a single linear segment in two dimensions. (a) The renderer takes as input a set of PSFs corresponding to a discrete set of point source locations \((X, Z)\). To generate an image of the line segment (shown in red), the system first approximates the PSFs at the vertices \((X^v, Z^v)\) by spatial interpolation (b). Then, it interpolates the PSF at every point on the line segment using the PSFs at the vertices. Finally, it sums the contributions from each point weighted by the texture (emitted radiance) at that point (c).
We can rewrite this as

\[ I(x) \approx \sum_v k(x; X^v, Z^v) * (w_v(x) T(x)), \]  

(2.2)

where \( T(x) \) denotes the texture at the point that projects to \( (x, 1) \) on the image plane. The approximation approaches equality if we specify the PSFs more finely, i.e., use a finer sampling of locations \((X_k, Z_k)\), and if we use finer step sizes for \( v \) in the sum.

The process as described so far applies to any continuous surface. When we encounter discontinuities in the surface depth, the renderer must make some approximations. In reality, depth discontinuities create occlusion events, which refers to regions that are not visible to the full lens aperture—if we performed ray tracing, rays from these occlusion boundaries would intersect with only part of the aperture. These effects are difficult to render using non-physics-based techniques, but the following method gives a close approximation.

We divide the discontinuous scene into continuous linear segments \( l = 1, \ldots, N \), and for each segment \( l \) we separately render its image \( I_l(x) \) using equation (2.2). Additionally, we render a blur mask \( M_l(x) \) that is an image of segment \( l \) with constant texture \( T(x) = 1 \). To form the completed image, we sum all the images of the segments together, multiplied by the blur masks of the foreground segments:

\[ I(x) = \sum_l I_l(x) \prod_{s \text{ occludes } l} (1 - M_s(x)). \]  

(2.3)

See Figure 2.3 for an example of this scheme put in practice for a sample scene.

For training the Focal Track and metalens systems, Guo et al. use this renderer to generate a dataset of 500 tuples \((I_+, I_-, Z_{\text{true}})\) of randomly-generated two-layer scenes. The images are comprised of flat planes with randomized slants and tilts and randomly-selected shapes and textures taken from the COCO database. Each depth \( Z_{\text{true}} \) is also randomly generated and passed as input into the renderer.

While the metalens renderer suffices for training Focal Track and the metalens depth sensor, its
Figure 2.3: The metalens renderer generates images of the foreground $I_0$ and the background $I_1$ separately using equation (2.2) and combines them together using the masks $M_0$ and $M_1$ according to equation (2.3). This scheme closely approximates the scene at the depth boundaries.

...scope is limiting in terms of its physical accuracy, especially at occlusion boundaries, and the types of scenes it can generate. If our goal is to build systems that can produce high-density and high-accuracy depth maps, we need a more physically accurate renderer that can generate more realistic scenes beyond flat overlapping planes.

2.2 Validation Renderer

To help me check the validity of my Mitsuba-rendered images, I built a simple convolution-based renderer that simulates defocus for single-layer fronto-parallel scenes. This renderer assumes the thin-lens model for the camera (see Figure 2.4) with an aperture diameter $D$, focal length $f$, in-focus object distance $Z_f$, sensor distance $Z_s$, and a square sensor with width $w$ and a resolution of $N$ by $N$ pixels. Given this model, if we place a textured surface in front of the camera at a distance $Z$, the camera parameters and object distance determine the amount of blur we will see in the captured image. More specifically, the resulting blurred image $I(x, y)$ can be written as a pillbox blur kernel...
Figure 2.4: Thin lens camera model with aperture diameter $D$, focal length $f$, in-focus object distance $Z_f$, sensor distance $Z_s$, and sensor width $w$.

$k(x, y, Z)$ convolved with the all-in-focus pinhole image $P(x, y)$:

$$I(x, y) = k(x, y, Z) \ast P(x, y),$$  \hspace{1cm} (2.4)$$

where $(x, y)$ are pixel coordinates and the size of kernel $k(x, y, Z)$ is determined by the camera geometry and depth $Z$ of the surface. For the thin-lens model, we can compute the size of $k(x, y, Z)$, which in optics is called the circle of confusion, using similar triangles.

Figure 2.5 shows a diagram for this calculation. Using similar triangles, we get the relation

$$\frac{y}{D/2} = \frac{|Z - Z_f|}{Z}. \hspace{1cm} (2.5)$$

Additionally, we have

$$\frac{y}{c/2} = \frac{1}{m}, \hspace{1cm} (2.6)$$

26
Figure 2.5: A point at depth $Z$ gets blurred into a finite width $c$ at the image plane. We can use similar triangles to write the size of $c$, also called the circle of confusion, in terms of the camera parameters.

where $c$ is the size (diameter) of the circle of confusion in the world coordinate system and $m = \frac{Z_0 - f}{f}$ is the magnification of the lens, which relates the size of objects at the in-focus distance to the size of their images on the photosensor. Combining equations (2.5) and (2.6) to solve for $c$, we get

$$c = mD \left| \frac{Z - Z_f}{Z} \right|.$$  \hspace{1cm} (2.7)

Equation (2.7) gives us the size of the blur kernel in world coordinates. Our kernel $k(x, y, Z)$ is in pixel coordinates, so to compute its radius—call it $r(Z)$—we need to convert between the units by dividing $c$ by the sensor pixel size $w/N$, which yields

$$r(Z) = \frac{c}{w/N} = \frac{mD |Z - Z_f| \cdot N}{Z \cdot w}.$$ \hspace{1cm} (2.8)
Lastly, for a camera, we usually know parameters $D, Z_f, Z_s, w,$ and $N,$ but not $f$ or $m.$ We can rewrite $r(Z)$ in terms of the known parameters. First, recall the thin lens equation:

$$\frac{1}{Z_s} + \frac{1}{Z_f} = \frac{1}{f}.$$  

(2.10)

This formula allows us to rewrite the magnification as $m = \frac{Z_s}{Z_f}$ and the blur kernel radius becomes:

$$r(Z) = \frac{Z_s \cdot D \cdot |Z - Z_f| \cdot N}{Z_f \cdot Z \cdot w}.$$  

(2.11)

The blur kernel $k(x, y, Z)$ is simply a $(2r-1)$ by $(2r-1)$ matrix whose elements are 1 in a circle of radius $r(Z)$ and 0 everywhere else. Now, given a camera with specified parameters, a planar surface, and a distance $Z,$ we can render physically accurate images with equations (2.4) and (2.11).

2.3 Mitsuba Renderer

Mitsuba is an open-source physics-based rendering software that promises high-quality, photorealistic data. My goal was to use Mitsuba to generate physically accurate defocused images and accompanying depth maps given a set of camera parameters and a digital description of a scene. Here, I describe how I set up Mitsuba as a renderer for generating depth-from-defocus data, and I show some experiments and results validating the fidelity of the renderer to real-world physics.

2.3.1 Building Blocks

Before trying to generate defocused scene images and depth maps, I first tried rendering arbitrary scenes to explore Mitsuba’s basic functions. I learned that the building blocks of a Mitsuba scene are (1) the emitter, (2) the integrator, (3) the sensor, and (4) shapes. Scene descriptions are written in XML, and might look like the following:
Emitters are equivalent to light sources. Mitsuba has several different options, including point light sources, directional light sources, area light sources (which turn geometric shapes into light emitters), spot lights, and more.

Integrators represent different approaches for computing the integrals for light transport. One setting for integrators is path depth, which refers to the number of times a light ray scatters between the light source and the camera. A smaller maximum path depth means faster rendering times but a less accurate resulting image, whereas larger path depths lead to longer rendering times but a more accurate final image. Some different integrators include the path integrator and the direct integrator, both of which are variations of the ray tracing algorithm and have minor differences in their sampling methods.

Sensors allow us to choose the type of camera capturing the scene and specify the associated parameters. One type of sensor is the perspective pinhole camera, which allows users to specify a view-
ing direction, a field of view, focal length, and maximum/minimum visible distances. Another sensor important for simulating defocused images is the perspective camera with a thin lens, which has additional parameters of aperture radius and focus distance. Within the sensor, we can also set values for a “film,” which contains information about the output image’s size and resolution.

Shapes define the objects in the scene. There are basic shapes included in Mitsuba, like disks, rectangles, cylinders, and spheres, but it is also possible to insert custom shapes by inputting OBJ or PLY files containing 3D object specifications. We can also apply texture patterns from any image source to shapes.

2.3.2 Pinhole Image

After learning Mitsuba’s basic functionality, I experimented with rendering all-in-focus pinhole images using a given set of camera parameters as an intermediate step towards rendering defocused images. As validation, I rendered a pinhole image using the metalens renderer with one set of camera parameters, and I used this as a baseline while I attempted to render the same image using Mitsuba with the same set of camera parameters.

The test image was a textured triangle in the foreground with a textured square in the background, both shapes in fronto-parallel orientation. In the metalens renderer, I created a triangle-shaped mask for the foreground, effectively choosing the location of the triangle on the image plane. The metalens renderer then selects depths for both the foreground and background. Next, using the known pixel coordinates of the vertices and the depth of each vertex, I used backprojection to compute the position of the triangle’s vertices in world coordinates.

With these coordinates, I created a 3D representation of the triangle stored in a PLY file that I passed in as a shape within a Mitsuba scene description. I used Mitsuba’s rectangle shape plugin for the background.

The first challenge of using Mitsuba is translating a set of real-world camera parameters into pa-
Figure 2.6: Mitsuba allows users to specify an aperture radius $r$, an in-focus distance $Z_f$, a field of view $\theta$, and a sensor resolution $N$. Nonadjustable parameters have been grayed out.

parameters for a Mitsuba sensor. Recall that for our real-world camera, we know camera parameters $D$, $Z_f$, $Z_s$, $w$, and $N$ as defined in section 2.2. (For simplicity, we continue to use a square image sensor with resolution $N$ by $N$.) For Mitsuba, we are allowed to specify an aperture radius $r$, an in-focus distance $Z_f$, a field of view $\theta$, and a sensor resolution $N$. (See Fig 2.6 for a diagram.) Fortunately most of the parameters overlap and it is not too difficult to translate from one system to the other to generate a pixel-perfect match between the metalens renderer’s output and Mitsuba’s output:

\[ r = \frac{D}{2} \]  \hspace{1cm} (2.12)

\[ Z_{f, \text{Mitsuba}} = Z_{f, \text{metalens renderer}} \]  \hspace{1cm} (2.13)

\[ \theta = 2 \arctan \left( \frac{w/2}{Z_s} \right) \]  \hspace{1cm} (2.14)

\[ N_{\text{Mitsuba}} = N_{\text{metalens renderer}} \]  \hspace{1cm} (2.15)
2.3.3 Depth Recovery

Another function we need for our data generation pipeline is the ability to measure and report depths given a 3D scene description. There are several ways to accomplish this task in Mitsuba. One can write a custom rendering pipeline using Mitsuba’s Python interface that measures depths while performing ray tracing, or alternatively, one can use Mitsuba’s built-in arbitrary output variable (AOV) integrator where one of the output variables is depth. Having tried both solutions, the solution using the AOV integrator is simpler in terms of implementation while also reporting more accurate depth values.

One idiosyncrasy of Mitsuba is that it reports depth relative to the camera center—that is, for each point in the world frame, it measures the distance from that point to the camera center rather than to the plane in which the camera center lies. As a result, a fronto-parallel scene with a constant $Z$ under our normal convention appears to be closest in the center and radially further towards the edges. But we can fix this and convert these radial depth measurements into conventional depth measurement with some geometry and trigonometry. (Figure 2.7 shows the before and after of the conversion.)

![Figure 2.7: Both images are depth maps of the same front-parallel scene where $Z = 0.434$. Mitsuba reports depth radially (left). However, we can use geometric relations to convert it back into a constant depth (right).](image)
2.3.4 Accurate rendering with thin lens

Having mastered pinhole images and depth recovery, we now turn to the task of rendering defocused images with Mitsuba’s thin lens. In my initial attempts at simulating defocus, I began with a single fronto-parallel plane with the goal of capturing an image that matched what I could generate with my physically-accurate validation renderer. (with the same set of camera parameters for each system)

At first it seemed that Mitsuba was missing degrees of freedom. To specify the amount of blur in the validation renderer, we use equation (2.11), which uses parameters $Z_s$, $Z_f$, $w$, and $N$. This means by changing the camera geometry, specifically the sensor distance $Z_s$ or the sensor width $w$, we can change the amount of blur in the captured image. However, Mitsuba has no settings for $Z_s$ and $w$.

My first hypothesis was that Mitsuba uses fixed values for $Z_s$ and $w$, untouchable by the user. I designed two different experiments to try to uncover these hidden values, both based on trying to match the blur in an image generated by the validation renderer, where $Z_s$ and $w$ are known, to an image generated in Mitsuba. In the first experiment, I set up the same pinhole image in Mitsuba and in the validation renderer, I guessed values for $w$, passed each guess into my validation renderer, generated images using my validation renderer, and compared it with my Mitsuba-generated image. In the second experiment, I tried to gather more quantifiable information about $w$ using a similar methodology: I generated stacks of images for a range $w$ in the validation renderer and computed their loss against a Mitsuba-generated image to find the $w$ associated with the least loss value.

However, I quickly realized these experiments would not work because all the images for the varying $w$ were actually identical. It turns out that the size of the blur kernel in pixel units is independent of both $Z_s$ and $w$. We can show this mathematically by first writing $Z_s$ in terms of $\theta$ and $w$.

*The documentation for Mitsuba 2 says nothing about these values, and I also emailed the creators of Mitsuba, who replied suggesting I design some experiments to figure out their values.
by rearranging equation (2.14) into

\[ Z_s = \frac{w/2}{\tan(\theta/2)}. \]

Next, substituting this expression for \( Z_s \) into equation (2.11) reveals the independence of \( r(Z) \) on \( Z_s \) and \( w \):

\[
r(Z) = \left( \frac{w/2}{\tan(\theta/2)} \right) \cdot \frac{D|Z-Z_f| \cdot N}{Z_f \cdot Z \cdot w} \\
= \frac{D|Z-Z_f| \cdot N}{2 \tan(\theta/2) \cdot Z_f \cdot Z}.
\]

This result seemed to suggest that knowing both \( Z_s \) and \( w \) is not essential to generating a physically accurate image in Mitsuba. Indeed, we can show that as long as the ratio between \( Z_s \) and \( w \) remains the same with all other parameters held constant, the captured image too will be the same. (The proof of this claim is in the appendix.) And fortunately for us, the field of view \( \theta \) fixes the ratio between \( Z_s \) and \( w \), given by equation (2.14).

Using these new insights, I edited my validation renderer to generate images without \( Z_s \) and \( w \) as input but with \( \theta \) instead. I tested these on a simple checkerboard texture pattern, and while the results looked adequately similar for some values, they looked very different for others. See Figure 2.8 for additional details.

These findings launched a new set of experiments to determine the root cause of the difference. First I checked the shape of Mitsuba’s blur kernel. I used a pillbox blur kernel in the validation renderer, but if Mitsuba used something else, like a Gaussian blur kernel, perhaps it could be the source of the discrepancy. To isolate Mitsuba’s PSF and reveal the shape of the blur kernel, I created a scene with a tiny spherical light source in the center and rendered images with a few different focus distances. (The tiny spherical light source approximates a delta function, which, when convolved with a blur kernel, will reveal the shape of the blur kernel.) I simulated the same images using the
(a) Defocused images of fronto-parallel checkerboard pattern at varying depths with an in-focus distance of 0.2 meters. For \( Z = 0.19 \) and \( Z = 0.18 \), the images look mostly the same. However the difference between the two images at \( Z = 0.17 \) is apparent.

(b) If we take a cross-section of the \( Z = 0.17 \) images at \( x = 200 \) and compare them, we can see that the intensities are not in sync, confirming that the images are indeed different.

**Figure 2.8:** Initial comparisons between Python- and Mitsuba-generated defocus.
validation renderer. Both renderers had the same pillbox blur kernel.

The blur kernel was not the source of the discrepancy, so in my next experiment, I wanted to determine whether it was an issue with Mitsuba’s renderer or the validation renderer. I chose to use a white-to-black gradient and render an out-of-focus image using each renderer because the gradient is a simpler scene than the checkerboard pattern and we know the expected output—with extreme blur, the gradient should look like solid gray, meaning that as we go more and more out of focus, the scene should tend more and more towards a constant gray tone. After rendering a defocused image in each renderer with identical parameters, I noticed that the two images were indeed different, with the Mitsuba output closer to my expectation (see Figure 2.9).

If the issue was with the validation renderer, there were two causes I could think of. The first cause could be the way that the code handles convolutions. There are different conventions for handling boundaries and for cropping the final output; however, after testing all nine combinations, I concluded that this was not the cause. The second culprit could be edge effects, which is related to how the convolutions handle boundaries. In the validation renderer code, the convolutions use

\begin{figure}
\centering
\includegraphics[width=0.7\linewidth]{mosaic.png}
\caption{On the left are the rendered defocused images of the gradient. On the right is their pixelwise difference, which is non-zero.}
\end{figure}
symmetric boundary conditions, which is often a good approximation but not physically accurate, especially as the blur kernel gets larger. To get around this issue, I manually zero-padded the gradient, adding a thick black border around the image—this way, edge effects become negligible. Rendering this image produced identical results (see Figure 2.10).

Returning to the original checkerboard pattern, I added the same zero-padded border around this image and got them to match (see Figure 2.11). We can conclude from this set of experiments that the Mitsuba renderer works as expected: it uses a pillbox blur kernel, and given a set of camera parameters and a scene description, generates physically accurate images.
Figure 2.11: Exact match between the two outputs once the border is added. Diagrams include the two defocused outputs, the intensities at the $x = 200$ cross-section, and the source image.
In this thesis, I have demonstrated how to generate photorealistic images from a digital scene description using the Mitsuba rendering software. I also validated its fidelity to real life physics through a series of experiments and comparisons with outputs from both the metalens renderer my own custom simulator.

One present limitation that suggests future work is the ability to adapt to custom PSFs other than pillbox blur kernels. Further work is required to extend the pipeline to include user-provided
PSFs, which would begin by writing a custom sampler that samples according to the importance of each directional ray that reaches a pixel\textsuperscript{19}. This will further enhance its generalizability to encompass depth from defocus systems that have highly customized computational optics.

Additionally, thus far I have been manually setting scenes and rendering them one at a time. An important next step will be to automate this process and turn it into a tool that is easy to use, takes in custom parameters, and generates a lot of training data at once—and hopefully it can be built into a generalized tool for all researchers to use in the near future.

A more general future direction for this data generation pipeline is to provide high quality data for all kinds of different depth from defocus systems that need training data. Its efficiency will save researchers time and its fidelity to real-world physics will help improve the quality of new and existing models.

New robust datasets also have the potential to inspire and enable new architectures that will improve the tradeoff between the accuracy and density of depth from defocus systems. A possible architecture that could see improved results as a result of improved data is one that combines generative inpainting and gated convolutions\textsuperscript{25} with Guo et. al’s depth and confidence predictor\textsuperscript{9,10}. While normally meant for tasks such as removing watermarks or blemishes from photos, techniques within the inpainting network could be transferred to the task of filling in the empty areas of a thresholded depth map. With new and improved data, I am hopeful for this network to yield promising results. But this is just one of an endless number new possibilities. There are many more architectures that have yet to be developed and that could benefit from customizable and physically-accurate training data.
A.1 Converting Radial Depth to Conventional Depth

Consider the setup in Figure A.1. This is a 2D version of the 3D problem we want to solve, but once we solve this simpler version we can easily generalize to 3D. For any given point \( u \) on the sensor, where \( u = \frac{y}{w/2} \) is in pixel coordinates, there is a corresponding angle \( \alpha \) that determines the angle at...
which the radial depth $Z_R$ is measured. The depth we are looking for is

$$Z = Z_R \cos \alpha, \quad \text{(A.1)}$$

so it suffices to find a formula for $\alpha(u)$, because then we can compute $Z$ with (A.1).

First we can write down

$$\tan \alpha = \frac{y}{Z_s}. \quad \text{(A.2)}$$

From the definition of $u$, we can substitute $y = \frac{uw}{2}$:

$$\tan \alpha = \frac{uw/2}{Z_s}. \quad \text{(A.3)}$$
We also know from equation (2.14) that \( \frac{w}{Z_s} = \tan \frac{\theta}{2} \), which means

\[
\tan \alpha = u \tan \frac{\theta}{2},
\]

(A.4)

or

\[
\alpha(u) = \arctan \left( u \tan \frac{\theta}{2} \right).
\]

(A.5)

Using Equation (A.5), we can simply precompute \( \alpha(u) \) for every pixel \( u \) and multiply its associated \( Z_R \) measurement by \( \cos \alpha(u) \) to get \( Z_R(u) \cos \alpha(u) = Z(u) \). To generalize to 3 dimensions, we can use spherical coordinates and treat \( u \) as the radial distance, and then apply the same formula to recover \( Z \).

### A.2 Proof: Rendered Image Depends Only on Ratio Between Sensor Width and Sensor Distance

Consider the following setup in Figure A.2. We have two cameras, one with sensor width \( w \) and sensor distance \( Z_s \) and one with sensor width \( w' \) and sensor distance \( Z'_s \), where \( \frac{w}{Z_s} = \frac{w'}{Z'_s} = r \).

We want to prove that the image on each of these sensors is identical to the other. Another way to frame the question is, given a point \( h \) on an object at a distance \( Z \), will \( h \) be projected onto the same position on each photosensor (in pixel coordinates)? Let us define \( u = \frac{y}{w/2} \) and \( u' = \frac{y'}{w'/2} \). We want to show that \( u = u' \) and that they depend on \( r \) and not the individual values of sensor width and sensor distance.

We can first note that

\[
\frac{b}{Z} = \frac{y}{Z_s} = \frac{y'}{Z'_s}.
\]

(A.6)
Figure A.2: Setup for proof that the rendered image depends only on the ratio between sensor width and sensor distance.

From here, we can write \( u \) as:

\[
\frac{y}{w/2} = \frac{Z_s \cdot b/Z}{w/2} = \frac{2b}{Z} \cdot \frac{Z_s}{w} = \frac{2b}{Zr}.
\]  

(A.7)

Similarly,

\[
\frac{y'}{w'/2} = \frac{Z_s' \cdot b/Z}{w'/2} = \frac{2b}{Z} \cdot \frac{Z_s'}{w'} = \frac{2b}{Zr}.
\]  

(A.8)

Thus, \( u = u' \) and both depend on \( r \) instead of individual values of sensor width and sensor distance, which is what we wanted to show.
References


This thesis was typeset using \LaTeX, originally developed by Leslie Lamport and based on Donald Knuth’s \TeX. The body text is set in 11 point Egenolf-Berner Garamond, a revival of Claude Garamont’s humanist typeface. The above illustration, “Science Experiment 02”, was created by Ben Schlietter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD thesis with this look and feel has been released under the permissive MIT (x11) license, and can be found online at github.com/suchow/Dissertate or from its author, Jordan Suchow, at suchow@post.harvard.edu.