Rondestvedt, Trond Jacob

3D Oriented Deep Learning Based Weather Data Augmentation

Master's thesis in Computer Science
Supervisor: Mester, Rudolf
Co-supervisor: Karolius, Kristian
June 2022

Master's thesis

Rondestvedt, Trond Jacob

3D Oriented Deep Learning Based Weather Data Augmentation

Master’s thesis in Computer Science
Supervisor: Mester, Rudolf
Co-supervisor: Karolius, Kristian
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Preface

This master thesis is written at the Department of Computer Science at the Norwegian University of Science and Technology, and is written in collaboration with DNV. The thesis concerns the field of Computer Vision and the problem of synthetically expanding a dataset using weather simulations.

Some of the images in this report were taken from publicly available web cameras around Norway and are used with permission from the owners of the cameras. I would like to thank Lofotcamera.no for the Svolvær scene and Moldewebcams.net for the Molde scenes.

I also would like to thank my supervisors Rudolf Mester and Kristian Karolius. They have been incredibly helpful during the course of this thesis.

Trond Jacob Rondestvedt

13.06.2022, Trondheim

Abstract

In this thesis our goal is to be able to synthetically expand maritime datasets by simulating different weather conditions. We look into how some monocular depth estimation architectures perform on maritime images. The depth information gained from the depth estimation architectures is used in weather models to simulate different weather conditions. We also end up collecting a new dataset that contains a large amount of maritime images as well as a fair amount of different weather images. The dataset contains scenes in both clear and other weather conditions, which makes it useful to simulate on and compare results to real weather. We present two weather models, one for fog and one for snow. The models are rather simple, but prove to be effective at simulating certain fog and snow conditions.

Sammendrag

I denne oppgaven er målet å kunne syntetisk utvide maritime datasett ved å simulere forskjellige type værforhold. Vi ser på hvordan noen monokulære dybdeestimering arkitekturer presterer på maritime bilder. Dybdeinformasjonen som man får fra dybdeestimeringen blir brukt i vær modeller for å simulere forskjellige værforhold. Vi endte opp med å samle inn et nytt datasett som inneholder en stor mengde med maritime bilder. I tillegg til dette fikk vi samlet inn en god del bilder med forskjellige type værforhold. Datasettet inneholder både bilder av klarvær og værforhold som tåke, regn og snø. Dette er gunstig for å kunne simulere syntetiske værbilder og sammenligne de med ekte værbilder. Vi presenterer to værmodeller, en for tåke og en for snø. Modellene er ganske enkle, men viser seg å være effektive på å simulere visse tåke- og snøforhold.

Contents

1. Introduction

There has been a lot of developments in deep learning methods in recent years and that has resulted in significant performance increase for object detection, image classification and segmentation tasks. In deep learning it is not only important to have a large amount of data to use for training, it is also very important to have a significantly varied dataset.

This would mean that the dataset would not only need to contain enough object classes, but it would also need a varied amount of lighting and weather conditions present. To collect a dataset that fulfills the requirement of being varied enough as well as being big enough is very hard.

In this thesis we focus on maritime environments and are looking to synthetically expand maritime datasets with different simulated weather conditions. This will hopefully prove to be useful for training object detectors of ships, buoys and other maritime objects. We have collected our own dataset of maritime images which was needed in this thesis. This dataset contains clear weather images as well as a varied amount of weather conditions, for example fog, snow and rain. We then use monocular depth estimation architectures to estimate the depth in these images. That depth information is then used by weather simulation models to simulate fog or snow on a clear weather image.

2. State of the art

In this chapter we will go through some papers we consider relevant in this thesis. First we will go through the monocular depth estimation architectures that we've experimented with. After that we go through a set of papers on weather simulation that we consider relevant in this thesis.

2.1. Monocular Depth estimation

Monocular Depth estimation is the task of finding a depth value for each pixel in a single image. This is useful for different types of work in computer vision where 3D information is needed. They can estimate the depth in existing single images instead of needing to take depth measurements, for example using a LIDAR, or estimating with a stereo image pair. It is especially helpful for getting depth information from existing image datasets where depth data is not already known. The quality and consistency of monocular depth estimation architectures, especially in maritime scenes, is something we want to test.

Below are the monocular depth estimation architectures experimented with in this thesis.

3D Ken Burns depth estimation (Niklaus et al. 2019a)

In this paper, they aim to produce the effect they refer to as the 3D Ken Burns effect which involves virtual zooming and opening of still images. To achieve this goal they trained their depth estimation architecture on a depth dataset they created from virtual scenes. These virtual scenes varied in type and setting, for example indoor, urban, rural, and nature scenes. (Niklaus et al. 2019a)

In this thesis, we are mostly interested in their depth estimation method so when we refer to "3D Ken Burns" throughout this thesis we refer to their depth estimation model.

Niklaus et al. (2019): **"3D Ken Burns Effect from a Single Image"**

The Ken Burns effect allows animating still images with a virtual camera scan and zoom. Adding parallax, which results in the 3D Ken Burns effect, enables significantly more compelling results. Creating such effects manually is time-consuming and demands sophisticated editing skills. Existing automatic methods, however, require multiple input images from varying viewpoints. In this paper, we introduce a framework that synthesizes the 3D Ken Burns effect from a single image, supporting both a fully automatic mode and an interactive mode with the user controlling the camera. Our framework first leverages a depth prediction pipeline, which estimates scene depth that is suitable for view synthesis tasks. To address the limitations of existing depth estimation methods such as geometric distortions, semantic distortions, and inaccurate depth boundaries, we develop a semantic-aware neural network for depth prediction, couple its estimate with a segmentation-based depth adjustment process, and employ a refinement neural network that facilitates accurate depth predictions at object boundaries. According to this depth estimate, our framework then maps the input image to a point cloud and synthesizes the resulting video frames by rendering the point cloud from the corresponding camera positions. To address disocclusions while maintaining geometrically and temporally coherent synthesis results, we utilize context-aware color- and depth-inpainting to fill in the missing information in the extreme views of the camera path, thus extending the scene geometry of the point cloud. Experiments with a wide variety of image content show that our method enables realistic synthesis results. Our study demonstrates that our system allows users to achieve better results while requiring little effort compared to existing solutions for the 3D Ken Burns effect creation.

Towards Robust Monocular Depth Estimation: Mixing Datasets foir Zero-shot Cross-dataset Transfer - MiDaS 2.1 (Ranftl et al. 2020)

This paper experiments with ways to train robust monocular depth estimation models that need to perform in diverse environments. To train their dataset they use a mix of 4 existing datasets as well as a new dataset which they extracted from 3D movies (referred to as MIX 5). They report that their experiments advanced the state of the art in monocular depth estimation at the time. (Ranftl et al. 2020)

This depth estimation model will be referred to as "MiDaS 2" when it is used in this thesis.

Ranftl et al. (2020): ***Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer***

The success of monocular depth estimation relies on large and diverse training sets. Due to the challenges associated with acquiring dense ground-truth depth across different environments at scale, a number of datasets with distinct characteristics and biases have emerged. We develop tools that enable mixing multiple datasets during training, even if their annotations are incompatible. In particular, we propose a robust training objective that is invariant to changes in depth range and scale, advocate the use of principled multi-objective learning to combine data from different sources, and highlight the importance of pretraining encoders on auxiliary tasks. Armed with these tools, we experiment with five diverse training datasets, including a new, massive data source: 3D films. To demonstrate the generalization power of our approach we use zero-shot cross-dataset transfer, i.e. we evaluate on datasets that were not seen during training. The experiments confirm that mixing data from complementary sources greatly improves monocular depth estimation. Our approach clearly outperforms competing methods across diverse datasets, setting a new state of the art for monocular depth estimation.

Vision Transformers for Dense Prediction - DPT (MiDaS 3.0) (Ranftl, Bochkovskiy, and Koltun 2021)

This paper builds on MiDaS 2.1 with a new architecture that uses vision transformers instead of a convolutional network as a backbone. They use the dataset produced in the previous paper (MiDaS 2.1) extended with 5 additional datasets and refer to this combined dataset as MIX 6. They claim that MIX 6 is the largest training dataset (1.4 million images) for monocular depth estimation at the time of writing. (Ranftl, Bochkovskiy, and Koltun 2021)

We will refer to this depth estimation architecture as MiDaS 3 throughout this thesis.

Ranftl, Bochkovskiy, and Koltun (2021): ***Vision Transformers for Dense Prediction***

We introduce dense vision transformers, an architecture that leverages vision transformers in place of convolutional networks as a backbone for dense prediction tasks. We assemble tokens from various stages of the vision transformer into image-like representations at various resolutions and progressively combine them into full-resolution predictions using a convolutional decoder. The transformer backbone processes representations at a constant and relatively high resolution and has a global receptive field at every stage. These properties allow the dense vision transformer to provide finer-grained and more globally coherent predictions when compared to fully-convolutional networks. Our experiments show that this architecture yields substantial improvements on dense prediction tasks, especially when a large amount of training data is available. For monocular depth estimation, we observe an improvement of up to 28 percent in relative performance when compared to a state-of-the-art fully-convolutional network. When applied to semantic segmentation, dense vision transformers set a new state of the art on ADE20K with 49.02 percent mIoU. We further show that the architecture can be fine-tuned on smaller datasets such as NYUv2, KITTI, and Pascal Context where it also sets the new state of the art.

2.2. Weather simulation

To train good object detection architectures large and varied datasets are required. One of the challenges with collecting these datasets is to gather enough types of weather and lighting. Therefore the task of augmenting existing datasets with various types of simulated weather has become popular. There are a few approaches to the task of weather simulation, but in this thesis we focus on physics based modeling. Below is a selection of these papers that we consider relevant for this thesis.

Towards Simulating Foggy and Hazy Images and Evaluating Their Authenticity (Zhang, Zhang, and Cheng 2017)

This paper proposes a model to simulate fog at any elevation in an image. To do this they estimate the elevation of each pixel instead of using depth maps. They also use 3D Perlin noise to end up with more natural-looking results. They also design an authenticity evaluator for synthetic foggy images to objectively measure the simulation results. (Zhang, Zhang, and Cheng 2017)

Zhang, Zhang, and Cheng (2017): **"Towards Simulating Foggy and Hazy Images and Evaluating Their Authenticity"**

To train and evaluate fog/haze removal models, it is highly desired but burdensome to collect a large-scale dataset comprising well-aligned foggy/hazy images with their fog-free/haze-free versions. In this paper, we propose a framework, namely Foggy and Hazy Images Simulator (FoHIS for short), to simulate more realistic fog and haze effects at any elevation in images. What's more, no former studies have introduced objective methods to evaluate the authenticity of synthetic foggy/hazy images. We innovatively design an Authenticity Evaluator for Synthetic foggy/hazy Images (AuthESI for short) to objectively measure which simulation algorithm could achieve more natural-looking results. We compare FoHIS with another two state-of-the-art methods, and the subjective results show that it outperforms those competitors. Besides, the prediction on simulated image's authenticity made by AuthESI is highly consistent with subjective judgements (Source codes are publicly available at https://github.com/noahzn/FoHIS).

Physics-Based Rendering for Improving Robustness to Rain (Halder, Lalonde, and Charette 2019)

In this paper, they propose a method to augment existing image datasets with realistic rain. Their method allows for setting the amount of rain to generate for an image. Rain that is determined to be too far away is rendered as a type of fog-like rain. They use a particle simulator to simulate the physics of the raindrops. Their method also allows for rendering fog with.(Halder, Lalonde, and Charette 2019)

Halder, Lalonde, and Charette (2019): **"Physics-Based Rendering for Improving Robustness to Rain"**

To improve the robustness to rain, we present a physically-based rain rendering pipeline for realistically inserting rain into clear weather images. Our rendering relies on a physical particle simulator, an estimation of the scene lighting and an accurate rain photometric modeling to augment images with arbitrary amount of realistic rain or fog. We validate our rendering with a user study, proving our rain is judged 40 percent more realistic that state-of-the-art. Using our generated weather augmented Kitti and Cityscapes dataset, we conduct a thorough evaluation of deep object detection and semantic segmentation algorithms and show that their performance decreases in degraded weather, on the order of 15 percent for object detection and 60 percent for semantic segmentation. Furthermore, we show refining existing networks with our augmented images improves the robustness of both object detection and semantic segmentation algorithms. We experiment on nuScenes and measure an improvement of 15 percent for object detection and 35 percent for semantic segmentation compared to original rainy performance. Augmented databases and code are available on the project page.

Simulating photo-realistic snow and fog on existing images for enhanced CNN training and evaluation (Von Bernuth, Volk, and Bringmann 2019)

This paper aims to simulate realistic snow and fog using depth information. The depth information they use is obtained from stereo image datasets. Using this depth information they reconstruct a 3D scene for the image in OpenGL. In the reconstructed 3D scene they distribute the snowflakes as a texture. The snowflakes are then rendered for several frames to simulate motion blur. They recognize that past a certain distance the snow particles would be no different from fog and only render snow particles for a certain distance from the camera. They use a fog algorithm to simulate the snow past that point. (Von Bernuth, Volk, and Bringmann 2019)

Von Bernuth, Volk, and Bringmann (2019): **"Simulating photo-realistic snow and fog on existing images for enhanced CNN training and evaluation"**

Verification and robustness testing of machine learning algorithms for autonomous driving is crucial. Due to the increasing complexity and quantity of those systems in a single vehicle, just driving the required distance with a newly developed vehicle is not feasible anymore: billions of hours on the street without failure are necessary to qualify for industry standards like ISO 26262. That is where simulation comes into play: machine learning algorithms are trained and evaluated on well known image data sets like KITTI or Cityscapes. But today's data sets mostly contain images taken under perfect weather conditions and therefore do not harden optical object detection algorithms against various weather conditions. This paper focuses on reusing these established and labeled data sets by augmenting them with adverse weather effects like snow and fog. Those effects are rendered physically correct and life like while being added to existing real world images. Thanks to easy parametrization the weather influences may be varied as necessary and allow for finely tuned learning and optimization processes. The weather effects are evaluated with regard to realism and impact on an established object detection algorithm. These newly created weather-influenced images may be used to validate or train new object detection algorithms.

Rendering Scenes for Simulating Adverse Weather Conditions (Sen, Das, and Sahu 2021)

In this paper, they propose a method to simulate fog and rain. They use both existing datasets with depth information and monocular depth estimation to obtain the depth information. The rainy scenes do not take into account the depth information and it is rendered in front of the image, but they do take into account the transparency of the rain by blending the rain streaks with the original image. They use the grayscale inverse depth map to compute the density of fog in the image. After that, they use Perlin noise to simulate heterogeneous fog. (Sen, Das, and Sahu 2021)

Sen, Das, and Sahu (2021): **"Rendering Scenes for Simulating Adverse Weather Conditions"**

Most of the object detection schemes do not perform well when the input image is captured in adverse weather. Reason being that the available datasets for training/testing of those schemes didn't have many images in such weather conditions. Thus in this work, a novel approach to render foggy and rainy datasets is proposed. The rain is generated via estimation of the area of the scene image and then computing streak volume and finally overlapping the streaks with the scene image. As visibility reduces with depth due to fog, rendering of fog must take depth-map into consideration. In the proposed scheme, the depth map is generated from a single image. Then, the fog coefficient is generated by modifying the 3D Perlin noise with respect to the depth map. Further, blending the corresponding density of the fog with the scene image at a particular region based on precomputed intensities at that region. Demo dataset is available in this https://github.com/senprithwish1994/DatasetAdverse.

3. Datasets

In this thesis we experiment with augmenting images with weather and we need a dataset of images to both augment with weather and real weather images to compare the augmented images against. Some other things are also need to take into consideration. We are going to run monocular depth estimation on the images and we believe that images with more context clues (buildings, trees, docks etc.) would be better for this case. Since we are looking to synthetically expand maritime datasets, this naturally means that we are especially interested in images from maritime environments.

To summarize, we are looking for datasets with images from maritime scenes where varied weather conditions as well as clear weather conditions are present.

3.1. Existing datasets

Here we will take a look at a couple of datasets we considered, however, we did not end up using these datasets. The main reason for this is that we collected our own dataset but we will mention some of the problems with these existing datasets for use in this thesis.

COCO Dataset (Lin et al. 2014)

The COCO dataset is a large dataset with images of various objects intended for training object detection and segmentation (Lin et al. 2014). This dataset is focused on all types of objects. Boat is an object class included in that however most of this dataset does not consist of maritime scenes with boats. This is possible to confirm by searching for the keyword "boat" in their dataset. This search returns a result of 3146 results, which is a small part of the dataset considering the dataset consists of 123287 images (Lin et al. 2014). Another factor to consider is that after having a quick look through the relevant images from this dataset we did not find any images that contained any considerable weather. Below is a couple of example of maritime scenes from the COCO Dataset.

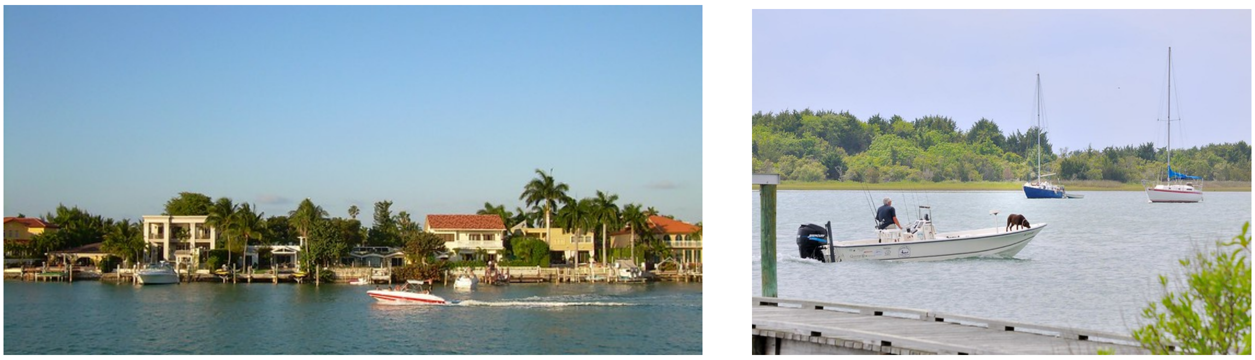

Figure 3.1.: Examples of maritime scenes in the COCO Dataset. (Lin et al. 2014)

Lin et al. (2014): ***Microsoft COCO: Common Objects in Context***

We present a new dataset with the goal of advancing the state-of-the-art in object recognition by placing the question of object recognition in the context of the broader question of scene understanding. This is achieved by gathering images of complex everyday scenes containing common objects in their natural context. Objects are labeled using per-instance segmentations to aid in precise object localization. Our dataset contains photos of 91 objects types that would be easily recognizable by a 4 year old. With a total of 2.5 million labeled instances in 328k images, the creation of our dataset drew upon extensive crowd worker involvement via novel user interfaces for category detection, instance spotting and instance segmentation. We present a detailed statistical analysis of the dataset in comparison to PASCAL, ImageNet, and SUN. Finally, we provide baseline performance analysis for bounding box and segmentation detection results using a Deformable Parts Model.

Seaships 7000 Dataset (Shao et al. 2018)

The Seaships dataset contains images of maritime scenes with annotation for training object detection models. The Seaships 7000 version of the dataset only contains a sample of 7000 images from the dataset. The images are taken from web cameras and would need to be cropped before running monocular depth estimation on it. Although there are weather images and clear weather images of the same scenes, there does not seem to be enough types of weather images in the Seaships 7000 portion of the dataset. Below is a couple of examples from the Seaships 7000 dataset.

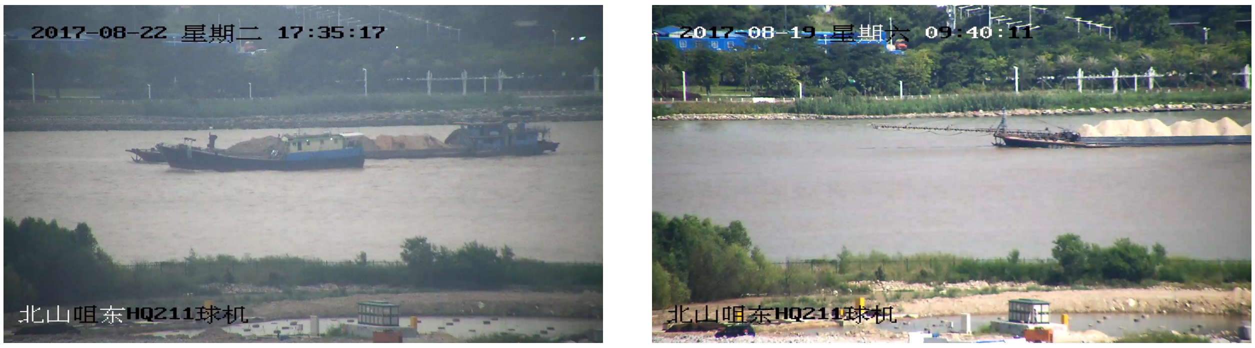

Figure 3.2.: Examples of images in the Seaships 7000 Dataset. The image on the left is taken in slightly foggy conditions while the one on the right is taken on a clear day. (Shao et al. 2018)

Shao et al. (2018): **“SeaShips: A Large-Scale Precisely-Annotated Dataset for Ship Detection”**

In this paper, we introduce a new large-scale dataset of ships, named as SeaShips, which is designed for training and evaluating ship object detection algorithms. The dataset currently consists of 31455 images and covers 6 common ship types (ore carrier, bulk cargo carrier, general cargo ship, container ship, fishing boat, and passenger ship). All the images are from about 10080 real-world video segments, which are acquired by the monitoring cameras in a deployed coastline video surveillance system. They are carefully selected to mostly cover all possible imaging variations, e.g., different scales, hull parts, illumination, viewpoints, backgrounds, and occlusions. All images are annotated with ship type labels and high-precision bounding boxes. Based on the SeaShips dataset, we present the performance of three detectors as a baseline to do the following: 1) elementarily summarize the difficulties of the dataset for ship detection; 2) show detection results for researchers using the dataset; and 3) make a comparison to identify the strengths and weaknesses of the baseline algorithms. In practice, the SeaShips dataset would hopefully advance researches and applications on ship detection. OAPA

3.2. Collecting a new dataset

By collecting our own dataset we can make sure that it fulfills all of the conditions that we need for it. Like mentioned above, we need a dataset containing maritime scenes with varied weather conditions. We considered two approaches for collecting the dataset.

At first it was considered to construct a camera rig with stereo cameras. With a stereo rig it would both be possible to capture images of different weather conditions and produce a depth map of the scene. This would have the benefit of directly choosing the scene and time of capturing the images. That means that we would go and capture images of specific weather conditions that we needed when they were happening. The depth maps could also be used as a reference to get a measure of the quality of the monocular depth estimation architectures. However, due to complications with setting up the stereo rig and that the needed equipment potentially taking months to arrive, this approach was not possible.

The approach we used is based on collecting images from publicly available webcams around Norway. These web cameras are available from harbors around Norway as well as some individually managed web cameras. This means that it was possible to pick and choose the scenes that we believe would be best for our use case. Once the desired cameras had been chosen, a simple script was written to collect an image from each camera every hour. In order to have the best chance of getting the needed images of various weather conditions the script was kept running from January and through April. This also gave us valuable images of winter turning into spring and the turbulent weather that can occur during the transition between these seasons. The images we've used in the report is used with permission from the owners of the cameras. The Svolvær scene is used with permission from Lofotcamera.no and the Molde scenes is used with permission from Moldewebcams.net. Below you can see some examples of weather captured in the dataset.

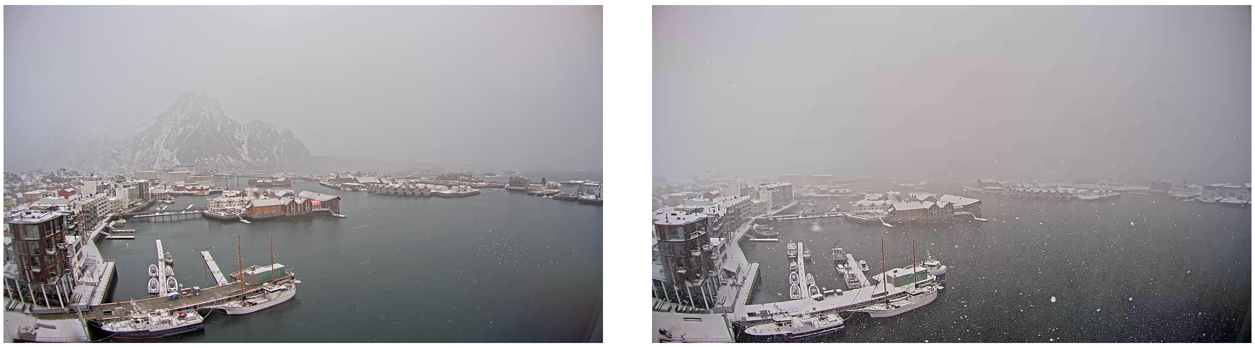

Figure 3.3.: Example of the Svolvær scene. On the left you can see an image with light snow and fog in the background. On the right you see heavier snow.

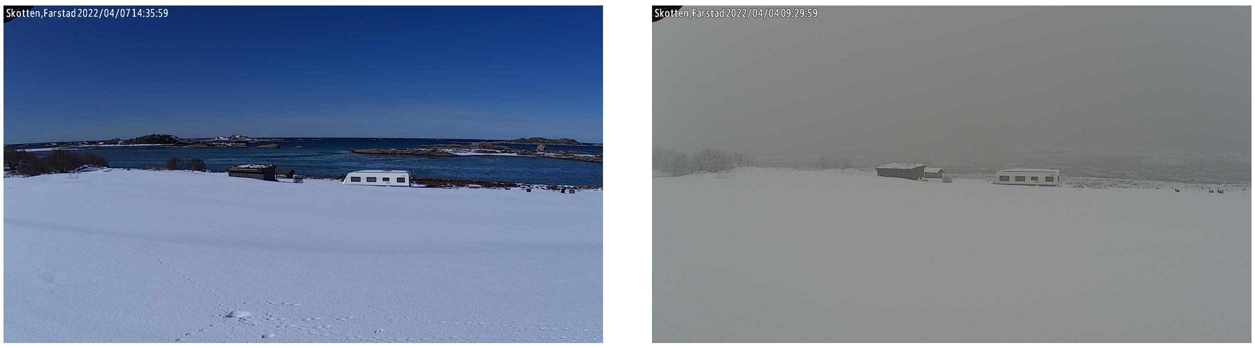
Skotten,Farstad 2022/04/07 14:35:59
Skotten,Farstad 2022/04/04 09:29:59

Figure 3.4.: Example of the Skotten scene. On the left you can see an image with clear weather. On the right you see an example of a foggy image.

We collected images from 25 maritime scenes around Norway. Some of the cameras stopped working after a while but from the cameras that kept working we collected over 3000 images of each scene. This includes night scenes that vary in appearence between the different scenes, as some of the cameras has a night mode which captures a grayscale image at night but others don't have this and captures color images.

In April, the time interval for downloading the images was reduced from an hour down to fifteen minutes to capture more images of weather and the transitions between different types of weather. Most of the cameras were set to only update the image every hour, so this was only possible to do for the cameras that updated often enough.

4. Monocular Depth estimation

The plan for this thesis is to utilize deep learning architectures to estimate the depth in single images. That information can then be used as a tool by the weather simulation models. Another thing to keep in mind is that the focus area of this thesis is simulating weather in maritime environments. So we experimented to see which of the depth estimation architectures is best for that case. Furthermore, this chapter shows our attempt at improving the depth estimation data and how we compared it to a manual depth map of the Svolvær scene using measurements from Google Maps.

4.1. Experimenting with depth estimation architectures

During this thesis we have experimented with three monocular depth estimation architectures. These architectures have trained on different datasets. The 3D Ken Burns depth architecture was trained on a variety of virtual scenes (Niklaus et al. 2019a). MiDaS 2 depth architecture was trained on a combination of 5 datasets from real world environments (Ranftl et al. 2020). Lastly, the MiDaS 3 architecture was trained on the same dataset as MiDaS 2 but with an additional 5 datasets from real world environments (Ranftl, Bochkovskiy, and Koltun 2021). Due to the size of both of the MiDaS datasets, we can see that the MiDaS depth architectures have been trained on a large amount of varied images. The dataset used for training 3D Ken Burns depth was available from their GitHub repository however it is not available anymore so we only have the descriptions of the download links to go on for information about their dataset (Niklaus et al. 2019b). These descriptions mention a variety of scenes, for example city, pirates, warehouse and western, so it is not clear exactly what each of them represents, however we do know that the dataset used for 3D ken Burns Depth is not extremely large (Niklaus et al. 2019b).

We wanted to know how how well these three depth estimation architectures performed on maritime scenes. To find this out we ran the depth estimation architectures on a lot of images and scenes from our newly created dataset detailed in chapter 3.2. The images chosen were from a variety of clear weather, ranging from a cloudy day with clear lines of sight to sunny blue skies. The resulting depth maps were subjectively evaluated for their accuracy by simply looking at the image and the resulting depth maps. Later on we did manually annotate one of the scenes to get a better sense of how we should expect the depth map to look. The subjective evaluation was deemed good enough since the depth maps were varied and most of them were not accurate at all.

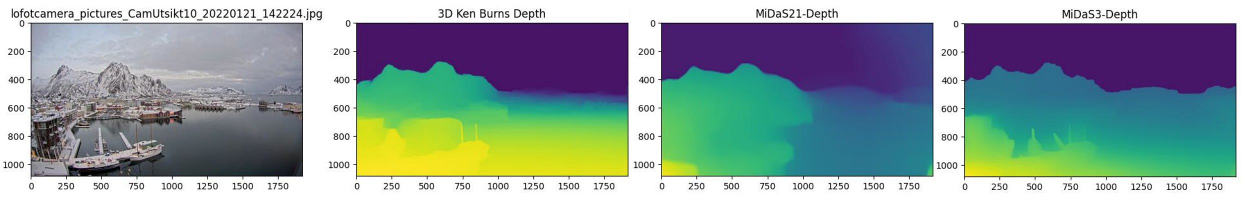

Figure 4.1.: On the left is the original image and each of the depth maps is noted with their corresponding architecture above each depth map. The image used in this figure is the one that 3D Ken Burns performed best on among all the images tested.

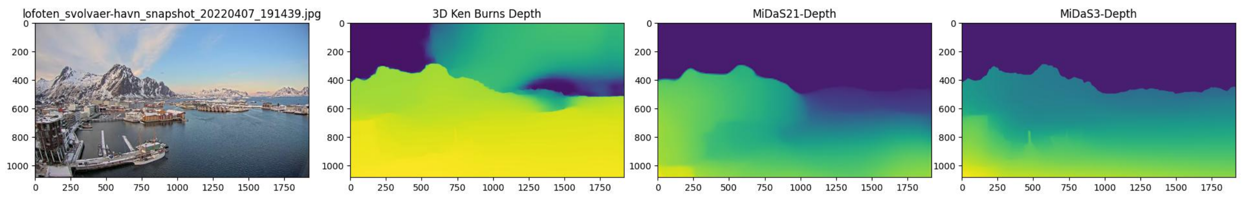

Figure 4.2.: On the left is the original image and each of the depth maps is noted with their corresponding architecture above each depth map.

The main theme from the evaluations were that 3D Ken Burns depth and MiDaS 3 depth performed better than MiDaS 2 depth. So it was decided to continue testing without MiDaS 2. After more testing, we noticed that while MiDaS 3 depth was far more consistent in its result in the various images and scenes tested while 3D Ken Burns provided the most accurate results overall. This result is shown in the figures above, where in figure 4.1 3D Ken Burns depth provides a pretty accurate depth map but in figure 4.2 it provides a bad result. Also notice that MiDaS 3 depth is consistent for the depth map for both images. This was reflected in the rest of the testing where most of the depth maps from 3D Ken Burns depth were somewhere in between the figures above (between accurate and bad depth maps), while MiDaS 3 depth kept more consistent but still had worse accuracy.

4.2. Normalizing the depth values

Up until this point we have only looked at the relative depth maps produced by each of the depth estimators, but the actual values of the depth maps vary a lot between the depth estimators. The MiDaS 3 depth architecture does not seem to predict depth in real distance values. It outputs values that vary from around 0 to 50 depending on the image and the scene, sometimes it even outputs negative values. On the other hand, 3D Ken Burns depth seems to output closer to real distance values and vary from around 0 up to and above 10000 depending on the image and scene. This difference in values does not necessarily mean that one depth estimation architecture is better than another. We only care if they are relatively accurate enough with its range of depth values, because then we can normalize the depth values to real distance values.

In our case, we have a collected a dataset where we effectively know the location of every camera and in which direction it is pointed. Therefore we can measure the distance from the camera to real world landmarks using Google Maps. This means that the simplest way to normalize the data would be to map the range from the smallest to the largest depth value to a new range with the corresponding distance values. By doing this we get better depth information by having it related to real distances. Without this we would have to guess at the maximum and minimum distances.

There is however another approach which we have experimented with. We know that the depth maps estimated by the depth estimation architectures are not completely accurate. This means that linearly normalizing the data, like explained above, would still have the same relative inaccuracies between the depth values across the image. Instead of doing it linearly, it should be possible to estimate a better function for normalizing the depth map from their relativistic values to real distance values. This could be done by sampling the real distance of more points across the image and use that to estimate a function. In this chapter we experiment with this possibility.

Collecting distance data from Google Maps

The first step to normalizing the depth data this way is to take the measurements from Google Maps. The process for doing this is very simple. Find the location of the camera. Look for recognizable landmarks in the image and match them with the same landmarks on the map. Then use the built in measuring tool in Google Maps to measure the distance to that landmark, then take note of the pixel coordinates and the the real distance to that pixel. This process still contains some estimates and would not provide exact distance but should still be a lot better than a random guess at the real distance.

Repeat this process for multiple points across the image and add in the depth estimations from the our three depth estimators we get a table as shown below.

x	y	true_distance	ken_burns_depth	midas21_depth	midas3_depth
330	1040	70	904.592	223.624	-0.80683
30	1060	70	892.1976	-3.68608	-1.62585
170	730	80	1126.547	281.9549	2.715733
920	890	85	1283.013	308.237	4.49149
800	650	210	1914.209	391.3132	9.168465
1350	600	340	3388.895	548.2685	11.30286
900	580	570	2256.416	407.1586	10.31755
600	560	650	2788.678	390.6364	10.18805
650	500	1500	3002.737	397.4896	10.76088
400	500	1500	2999.991	392.8285	10.14237
650	400	1500	3127.816	415.5565	11.03852

Table 4.1.: Here are real distances and depth measures by the 3 different depth estimators taken from an image in the Svolvær scene (lofotcamera_pictures_CamUtsikt10_20220209_145444.jpg).

Now that we have data points on distances for some pixels we extract the depth values for those pixels and plot the depth and distance values against each other for all the sampled pixels.

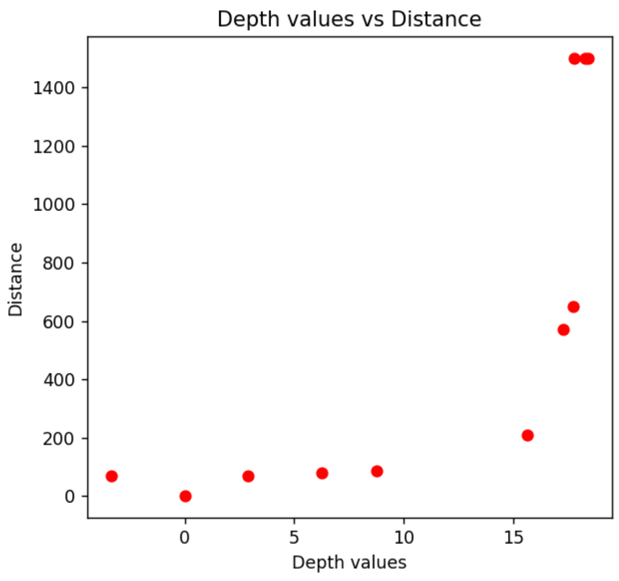

Figure 4.3.: Example of the points plotted with depth estimation from MiDaS 3 on the x-axis and measured real distance on the y axis.

In the figure above we noticed that as the depth estimation values increases, the real distance values increases exponentially. This gives us an idea of what type of function describes the relationship between these two sets of values. Another thing to notice in the figure is that we can see that there is no particular line or curve that perfectly fits all of the points. Therefore interpolation is not going to be able estimate a function for this relationship and we need regression or in other words we need to find the curve the best fits all of the points. We decided to use NumPy's polyfit to estimate the function, which is an implementation of least squares polynomial fit (NumPy 2022). The estimated function and the resulting normalization is shown in the figure below.

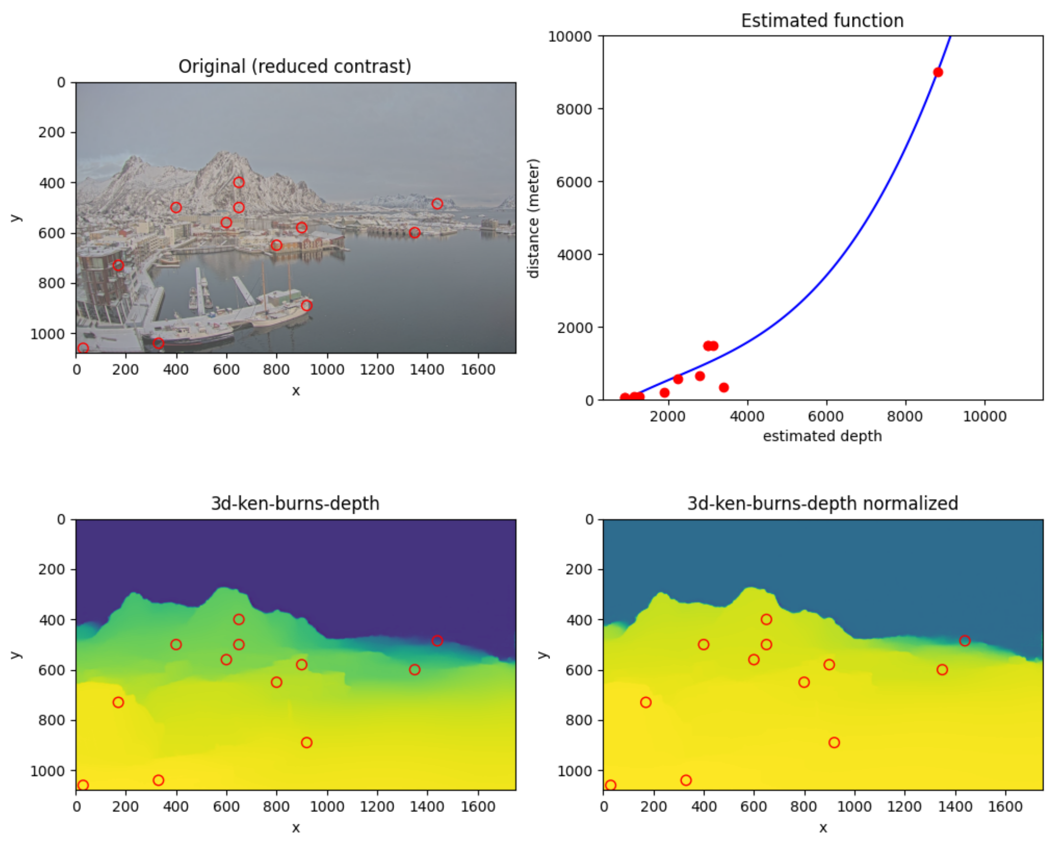

Figure 4.4.: Result when normalizing the depth data using NumPy's implementation of least squares polynomial fit.

The results from a few tests of this method was underwhelming and while we still believe this method might have some potential we decided to use the simple linear method to normalize from the depth values to real distance.

4.3. Manually annotating a depth map using Google Maps

Since we know the location of the cameras we can sample distance points and measure them using Google Maps. We do not need to limit this to only a few points and can do this for the whole image to create an estimated depth map. This is a very time-consuming process and is not worth the time if it needs to be done for a lot of different images. However, since the cameras we are using are statically mounted cameras it should be possible to do this process for a single image and it should be a pretty good estimation for that scene/camera.

To limit the amount of work needed to create one of these manual depth maps we first define a set of distance intervals. These intervals define how accurate the depth estimation is going to be so with that in mind we prioritize close distances. This is shown in the table below.

Distance (m)	**Color brightness**
10000	0
9000	9
8000	19
7000	28
6000	38
5000	47
4000	56
3000	66
2500	75
2000	85
1500	94
1000	103
900	113
800	122
700	132
600	141
500	150
450	160
400	169
350	179
300	188
250	197
200	207
150	216
100	226
50	235
0	255

Table 4.2.: This table shows the distance intervals chosen to annotate a depth map. The closer distance intervals are prioritized. For example, the color brightness of pixels within 0 m to 50 m would be 255, and the color brightness of pixels within 9000 m to 10000 m would be 9. Any pixels over 10000 m are 0, completely black.

Now that we have these distance intervals we can plot them as circles with a radius equal to each distance on Google Maps. We are using the Svolvaer scene for the visualizations.

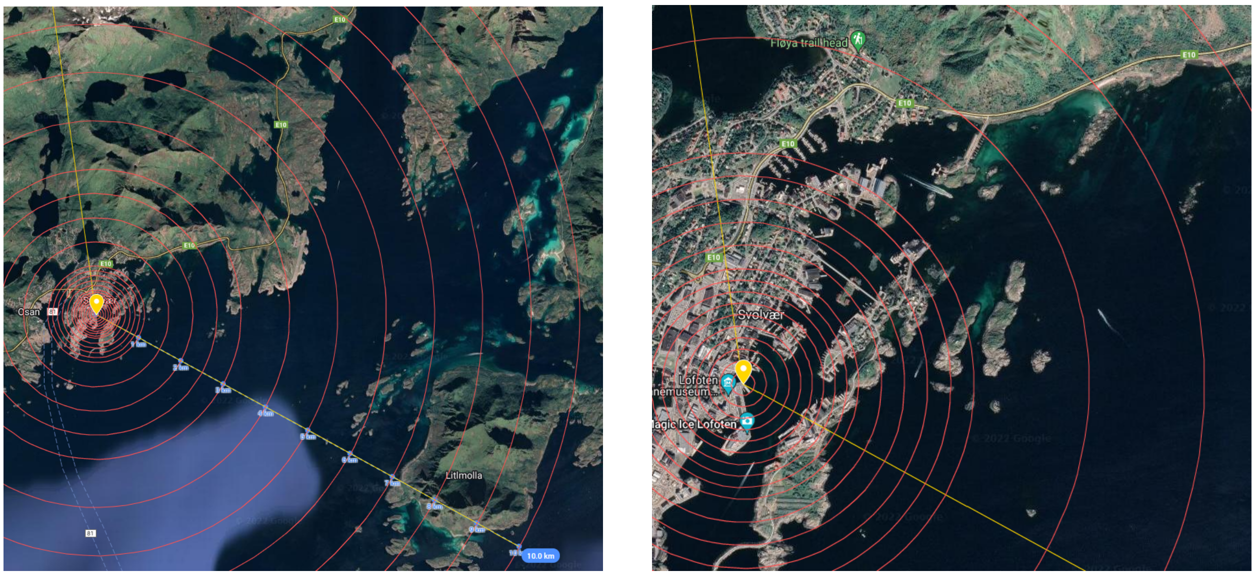

Figure 4.5.: On the images above you see the distance intervals visualized on Google Maps for the Svolvær scene. Each red circle represents a distance in the table. The yellow point is the camera's position and the yellow guidelines show the camera's approximate view of the scene. (Google 2022)

Next, is the time-consuming part, we need to match elements on the image to elements on Google Maps. We start by looking at one distance interval at a time and repeat the process until the entire image has been colored in with a color representing a distance interval. In this case, I started by isolating the sky and anything over a distance of 10 km and then moving down to the closest distance interval.

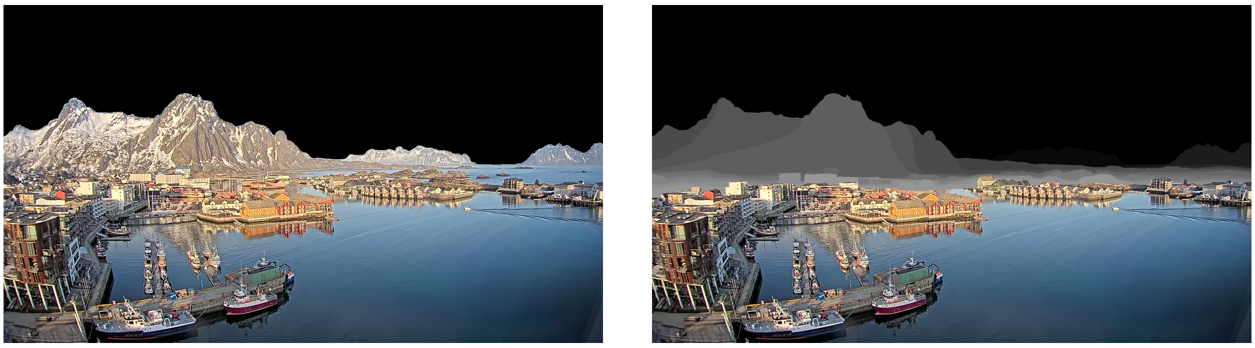

Figure 4.6.: On the left you see only one distance interval colored in (anything over 10 km), while on the right all the distance intervals down to 500 m are colored in.

This method is an estimate and not a true distance. That should be obvious by the fact that we are using distance intervals to annotate the depth, but it should also be noted that it is often hard to match elements on the image exactly to what is measured on Google Maps. As a result of this, we had to make guesses at several points in the image as to which distance interval elements fit. Once the entire image is colored in we have a greyscale image of the distance intervals as shown below.

With this greyscale image we can easily convert it into a depth map by converting each color brightness on the image to the distance it represents in table 4.2 above.

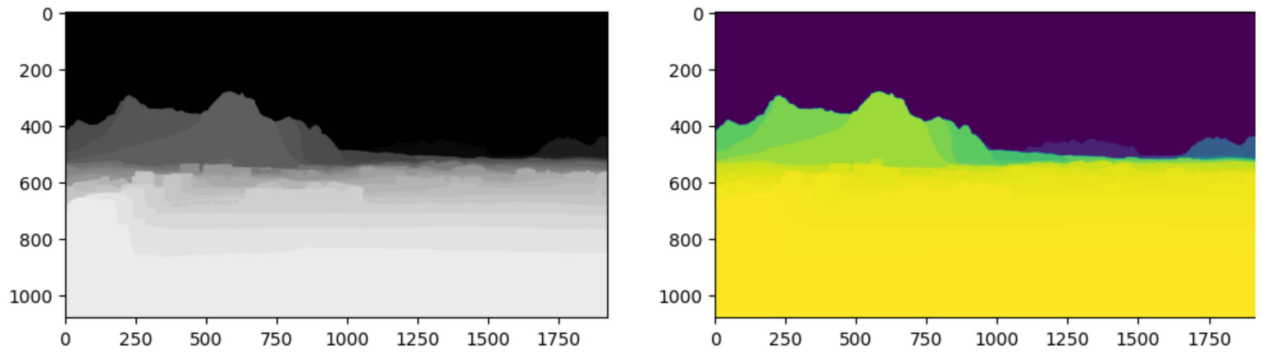

Figure 4.7.: On the left, the greyscale image is showing all the distance intervals drawn over the Svolvær scene. The black part is anything over 10 km while the brightest part is the closest distance interval, which is 50 m - 100 m in this case. On the right you see that image converted into a depth map using the reference table above, table 4.2.

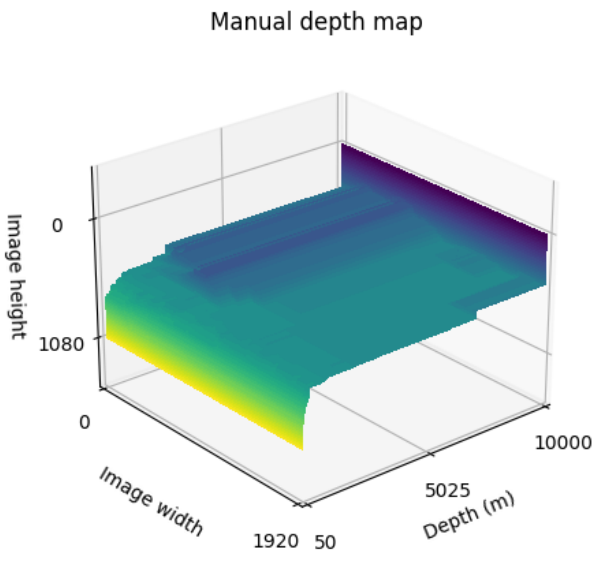

Figure 4.8.: This figure is showing the 3d surface plot of the Svolvær depth map.

5. Fog model

This chapter containing the fog model is largely based on my pre-project report (Rondestvedt 2021), chapter 3.

The fog model that is presented in this chapter is a simple physical fog model which is based upon how light interacts with the particles in the fog. We will describe the fog model in detail, explain how it was implemented in code and also discuss the limitations and problems with the model.

5.1. Theory

The fundamental idea of this fog model is to model how the light from the scene/image is changing as it passes through the fog. We look at the fog as divided into fog volumes with a length of Δ. As the light passes through these fog volumes, environmental light bounces of the particles in the fog. Some of the light that bounces of the particles hits the lens, which is called backscattering. This environmental light sort of illuminates the fog. So to summarize, the total light that reaches the camera through the fog is a mix of the attenuated light from the scene/image and the environmental light that backscatters of the particles in the fog. In the image below a single fog volume is visualize with what light goes into a fog volume and what light comes out.

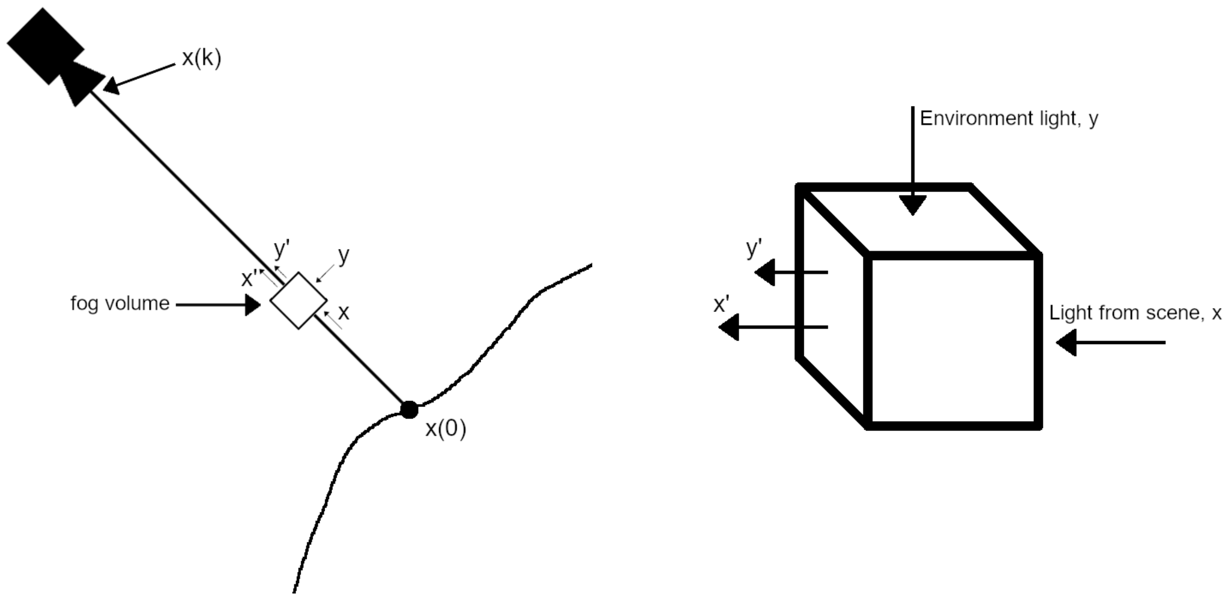

Figure 5.1.: The figure is showing how light passes through a fog volume and how environmental light backscatters off of the particles in the fog before reaching the camera sensor. Here only one fog volume is depicted, in reality there exist a lot of fog volumes between the scene and the camera sensor. x(0) is the original pixel light while x(k) is the pixel light after it has passed through k fog volumes.

The light that enters a fog volume, x, is attenuated by some factor α, while the environmental light, y, backscatters off the particles in the fog with a factor of β. This means that the attenuated light from the scene is $x' = \alpha \cdot x$ where $0 \leq \alpha \leq 1$, and the backscattering from environment light $y' = \beta \cdot y$ where $0 \leq \beta \leq 1$.

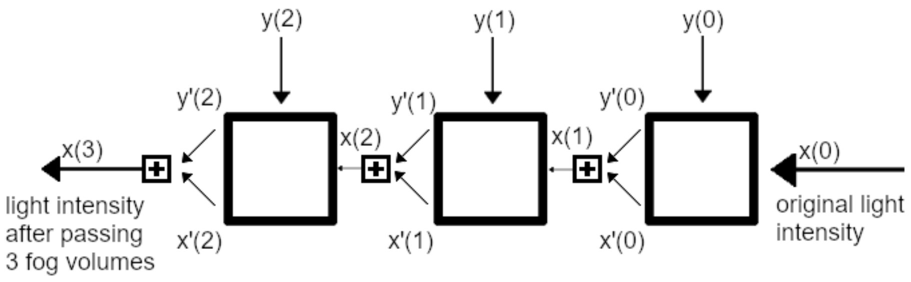

Figure 5.2.: Method for calculating light intensity through fog. x(0) is the original light intensity from the scene. The y(k) values are the environmental light that is reflected off the particles in the fog and backscattering. x(3) is the light intensity after passing through 3 fog volumes.

With this information and figure 5.2, we can start to define an equation for the augmented pixel light, $x(k)$. The equation for $x(k)$ as a function of k volumes of fog is:

$$\begin{aligned} x(k) &= x'(k-1) + y'(k-1) && (5.1)\\ &= \alpha \cdot x(k-1) + \beta \cdot y(k-1) && (5.2) \end{aligned}$$

if we assume that $y(k)$ are all equal ($y(k) = y_e$), we get:

$$\begin{aligned} x(k) &= x'(k-1) + y'_e && (5.3)\\ x(k) &= \alpha \cdot x(k-1) + \beta \cdot y_e && (5.4)\\ x(1) &= \alpha \cdot x(0) + \beta \cdot y_e && (5.5)\\ x(2) &= \alpha \cdot (\alpha \cdot x(0) + \beta \cdot y_e) + \beta \cdot y_e && (5.6)\\ &= \alpha^2 x(0) + \alpha\beta \cdot y_e + \beta \cdot y_e && (5.7)\\ x(3) &= \alpha^3 \cdot x(0) + \alpha^2\beta \cdot y_e + \alpha\beta \cdot y_e + \beta y_e && (5.8)\\ &\vdots \\ x(k) &= \alpha^k \cdot x(0) + \alpha^{k-1}\beta \cdot y_e + \alpha^{k-2}\beta \cdot y_e + \alpha^{k-3}\beta y_e + ... && (5.9)\\ x(k) &= \alpha^k \cdot x(0) + \beta \cdot y_e \cdot (\alpha^{k-1} + \alpha^{k-2} + \alpha^{k-3} + ...) && (5.10)\\ x(k) &= \alpha^k \cdot x(0) + \beta \cdot y_e \cdot \sum_{n=0}^{k-1} \alpha^n && (5.11)\\ x(k) &= \alpha^k \cdot x(0) + \beta \cdot y_e \cdot \frac{\alpha^k - 1}{\alpha - 1} && (5.12) \end{aligned}$$

We want the augmented pixel light as a function of distance, so we define $k = \frac{d}{\Delta}$ and get:

$$x(d) = \alpha^{\frac{d}{\Delta}} \cdot x(0) + \beta \cdot y_e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \tag{5.13}$$

Since β and y_e always appear as a product, we can simplify by defining $e = \beta \cdot y_e$:

$$x(d) = \alpha^{\frac{d}{\Delta}} \cdot x(0) + e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \tag{5.14}$$

Equation 5.14 defines $x(d)$ as a function of α, e and Δ. The α value defines how much of the original pixel light, $x(0)$, is retained when it passes through each fog volume before it reaches the final augmented pixel light, $x(d)$. While the e value defines how much environmental light is added for each fog volume. The last value, Δ, defines the length of each of the fog volumes. We want to set this Δ value so that we have enough fog volumes.

The equation does not take into account how the fog volumes can have different values of α, which could happen if the fog would have different densities, for example at different heights or over water or land. For this reason, the fog produced by this equation would end up being homogenous.

The augmented light from the fog volume is a sum of the attenuated light from the scene and the added light from the environment, this is shown in equation 5.13. We can argue that the values α and β are dependent on each other and add up to 1. We can use this in equation 5.13 and replace β with $1 - \alpha$ to get:

$$\begin{aligned}
\alpha + \beta &= 1 \\
\beta &= 1 - \alpha \\
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0) + (1 - \alpha) \cdot y_e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \qquad (5.15)
\end{aligned}$$

With this equation we have defined $x(d)$ as a function of α, y_e and Δ. The difference from equation 5.14 is that we now have both parts of the equation dependent on the α value. The α value still defines how much of the original pixel light is retained, but now it also defines of much environmental light can be added in the second part of the equation. We also have the y_e value to define how strong the environmental light added to the scene is.

5.2. Limiting cases

Here we will briefly look at the limiting cases for the fog model.

Case: $\alpha \to 0$

The α value controls how much of the original pixel light is retained for each fog volume. Therefore if this value goes towards 0 then less and less of the original pixel light will remain in the final augmented image. When the value reaches 0, none of the original pixel light will remain. Everything that appears in the augmented image will then be coming from the environmental light, y_e.

We can see this by setting $\alpha = 0$ in equation 5.15:

$$
\begin{aligned}
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0) + (1-\alpha) \cdot y_e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \\
x(d) &= 0 \cdot x(0) + 1 \cdot y_e \cdot \frac{-1}{-1} \\
x(d) &= y_e
\end{aligned}
$$

Case: $\boldsymbol{\alpha \rightarrow 1}$

As mentioned before, the α value controls how much of the original pixel light is retained for each fog volume. If this value raises, then more of the original pixel light will remain in the final augmented image (less fog). When the value is 1 then all of the original pixel light will remain (no fog). Since the β value is replaced with $1-\alpha$ in the equation 5.15 then no environmental light will be added. Therefore if $\alpha = 1$ the resulting image will be the same as the original image:

$$
\begin{aligned}
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0) + (1-\alpha) \cdot y_e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \\
x(d) &= 1 \cdot x(0) + (1-1) \cdot y_e \cdot \frac{1-1}{1-1} \\
x(d) &= x(0)
\end{aligned}
$$

Case: $\boldsymbol{y_e \rightarrow 0}$

In this case, less and less environmental light will be added to the augmented image. When $y_e = 0$ then no environmental light will be added to the image and the augmented image depends entirely on how much the original pixel light has been attenuated.

$$
\begin{aligned}
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0) + (1-\alpha) \cdot y_e \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \\
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0) + (1-\alpha) \cdot 0 \cdot \frac{\alpha^{\frac{d}{\Delta}} - 1}{\alpha - 1} \\
x(d) &= \alpha^{\frac{d}{\Delta}} \cdot x(0)
\end{aligned}
$$

Case: $\boldsymbol{y_e \rightarrow \infty}$

As the y_e value increases the augmented image will become brighter until it is completely white. With a high enough value of y_e, this will happen independently of how much of the original image is attenuated.

Case: $\boldsymbol{d \rightarrow \infty}$

As the distance to the original pixel increases, the more fog it passes through. The original pixel light is reduced more depending on α and more environmental light is added depending on the environmental light, y_e.

In this case, if $y_e = 0$, then the resulting image is completely black. The same as a night with no environmental light (no light from the moon or other sources).

However, if $y_e \neq 0$, then more and more light is added to the image as the distance increases which will result in a completely white image in the end.

5.3. Implementation

For the implementation of the fog model we use these parameters: an image array, an image depth array, γ, y_e, and Δ. The image array should contain all the pixels for the original image. The depth array should contain all the estimated depth values for each pixel in the image. The γ value converts into the α value from equation 5.15 like this; $\alpha = 1 - \gamma$. This means that a higher γ value results in a lower α value and a higher amount of attenuation of light from the original pixel (more fog). The y_e value sets the amount of environmental light to add to the image. The Δ value sets the length of the fog volumes, this value should probably remain constant at $\Delta = 1$.

Like we mentioned in chapter 4.2 the depth values given by the depth estimators are relative and not adjusted to real distance. This means that we have to normalize the depth values before using them in the fog model. We use the simple linear normalization method for this and normalize the depth values from the smallest to the largest values to 0 meters to 10 kilometers (the sky).

With our equation, the augmented pixel light could exceed 255 if the environmental light is set too high. This causes data overflow in the uint8 values for the image and causes glitches and mismatches in colors in the augmented image. To prevent this an uint16 augmented image array is first used, before trimming down the values to 255 and converting it back to uint8.

Speeding up the computation

The first implementation for this equation was using Python loops which was extremely slow. Each image augmentation took about 40-50 seconds to complete for an image of size 1280x720. This was way too slow to be able to efficiently experiment with values for augmenting the images.

To optimize the code we replaced the big for-loops and iterated over the arrays using universal functions in Numpy instead. This optimization reduced the time of the calculation to about 0.1 seconds per image. The total processing time (including loading and saving the image) was about 1 second for an image of size 1280x720. That is more than fast enough for the fog model.

The source code for this implementation of the fog model is given in appendix A.

5.4. Limitations with the fog model

There are two big limitations of the fog model. The first is that it can only produce homogeneous fog and the other is that the color of the fog is always grayscale.

The issue of homogeneous fog has already been solved in other papers by taking advantage of Perlin noise to adjust the depth map in order to simulate variable densities in the fog (Sen, Das, and Sahu 2021). This could be a possible improvement for this fog model as well.

Like mentioned above the other big limitation is that the fog is always a shade of gray. In real fog you can see that it can have a slightly different color depending on what time of day it is and other factors. Theoretically it should be possible to change the fog color by adjust the fog differently for the different color channels for each pixel, so that could be a solution to that problem.

6. Snow model

In this chapter we introduce a snow model that models snow using the pyramid view in front of the camera.

6.1. Theory

The fundamental idea of the snow model is to project the pyramid cone of the view in front of the camera and divide that volume into cells. These volume cells will then be filled with snowflakes using a binomial random generator based on the volume of the cell and the desired density of the snow.

6.1.1. Projecting the pyramid cone of the view

To project the pyramid cone of the view it is required to know the opening angles of the camera. In the dataset we've collected we do not know these parameters. The problem is that the captured images are from a lot of different cameras and the details of these cameras are unknown. Therefore the opening angle needed to be estimated. To make this estimate, we've looked at a selection of different cameras that is likely used in industrial scenes like the ones we have in the dataset. The table below shows these cameras and the details that the manufacturer listed for each of them. The different cameras were chosen based on similar resolution and what types of cameras were thought to be used to capture the images in our dataset.

Name	**Resolution**	**Focal length**	**Sensor size**	**Horizontal**	**Vertical**
Wisenet QNO-6022R	1920x1080	4 mm	1/2.8 inches	87.6°	
Wisenet QNV-6012R	1920x1080	2.8 mm	1/2.8 inches	113.7°	
Wisenet QNV-7012R	2560x1440	2.8 mm	1/3 inches	107.5°	
Wisenet QNV-7022R	2560x1440	4 mm	1/3 inches	78.2°	
AXIS M3116-LVE Network Camera	2688x1512	2.4 mm	1/2.7 inches	130°	72°
AXIS M3115-LVE Network Camera	1920x1080	2.8 mm	1/2.9 inches	105°	58°
AXIS P1375-E Network Camera	1920x1080	2.8 - 8 mm	1/2.8 inches	107-42°	57-24°
AXIS P1377 Network Camera	2592x1944	2.8 - 8 mm	1/2.7 inches	111-38°	81-28°
Ubiquiti UniFi G3-PRO	1920x1080	3 - 9 mm	1/2.8 inches	108-37°	58-20°
Ubiquiti UniFi G3-Bullet	1920x1080	3.6 mm	1/2.8 inches	85°	45°

Table 6.1.: This is a selection of different IP cameras that are used for surveillance in industrial settings. All the data is from the manufacturers of the cameras.

With the data in the table above, it is possible to calculate a horizontal opening angle, θ_w, using the focal length, f, and sensor size, w_s:

$$\theta_w = 2 \cdot \arctan\left(\frac{w_s}{2 \cdot f}\right) \tag{6.1}$$

The most common focal length, f, is 2.8 mm and the most common sensor size, w_s, is 1/2.8 inches ($\approx$ 9.071 mm) using the equation 6.1 this results in a horizontal opening angle, θ_w, of 116.6 degrees.

Even though the result for the horizontal opening angle is 116.6 degrees, the estimate does not need to be and probably is not accurate down to single degrees. Therefore we can round this down to an estimate of $\theta_w = 110$ degrees.

The resolution and the aspect ratio of the image are gathered from the images themselves and these will differ between some of the scenes. With this said, most of the cameras use a resolution of 1920x1080 and by extension an aspect ratio of 16:9. Therefore this resolution and aspect ratio will be used for the rest of the examples unless said otherwise.

Now that we have an estimate of the horizontal opening angle, θ_w, and an aspect ratio, R_a, we calculate the vertical opening angle, θ_h:

$$\theta_h = \frac{\theta_w}{R_a} \tag{6.2}$$

With the horizontal angle of $\theta_w = 110$ degrees this result in a vertical viewing angle of $\theta_h = 61.9$ degrees. The virtual camera characteristics we are working with are therefore:

Virtual Camera Characteristics	
Resolution	1920x1080
Focal length	2.8 mm
Sensor size	1/2.8 inches
Horizontal view angle	110 degrees
Vertical view angle	61.9 degrees

With this information, we can then project the pyramid view in front of the camera.

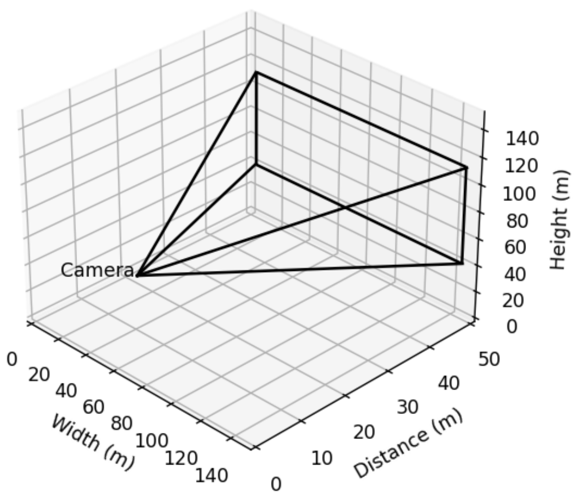

Figure 6.1.: This figure shows the pyramid view projected in front of the camera. The horizontal opening angle, θ_w, is 110 degrees and the vertical opening angle, θ_h, is 61.9 degrees. The pyramid view is projected up to 50 meters away from the camera.

6.1.2. Slicing the projected view and calculating the cell area

Once we have the projected view of the camera we slice it at certain distances. For each distance, d_f, we look at the volume between the front face area, a_f, and the back face area, a_{f+1}, at the next distance, d_{f+1}.

Front face area, a_f The front face area is the area sliced at the current distance, d_f.

Back face area, a_{f+1} The back face area is the area sliced at the next distance, d_{f+1}

Cell length, ℓ_c The cell length is the distance between the front face area and the back face area at each distance.

To start with we slice the pyramid view at constant increments of the cell length, ℓ_c. The slicing of the pyramid view is visualized in the figure below where the cell length, ℓ_c, is set to 5 meters. The front and back face area at a distance of 25 meters is highlighted in blue and red respectively.

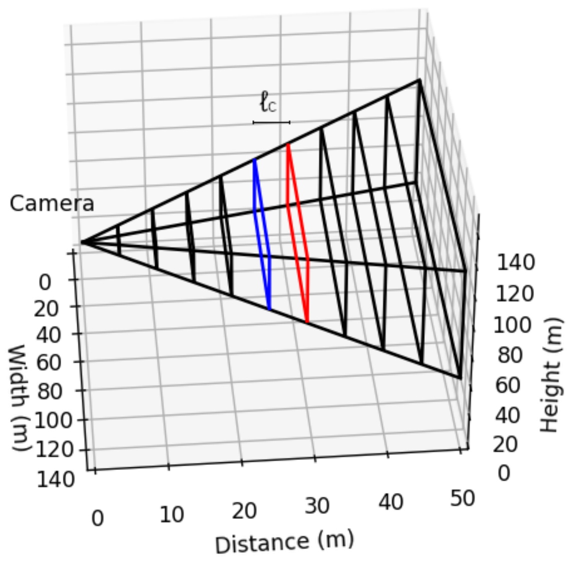

Figure 6.2.: Visualization of the pyramid view projected into 3D space and sliced at certain distances to find the area at those distances. This figure shows all the areas sliced with cell length, $\ell_c = 5$, from the camera to 50 meters away. The view is projected with a horizontal viewing angle of 110 degrees, an image size of 1920x1080 pixels.

The cell length of 5 meters was chosen to give the best overview, however going forward a cell length of 1 meter is used, $\ell_c = 1$.

For each distance, we need to calculate the sliced front and back face areas at each distance. The area is calculated the same for both faces so only the calculation of the front face will be shown. First, we find the width of the front face, w_f.

$$w_f = \tan\left(\frac{\theta_w}{2}\right) \cdot 2 \cdot d_f \tag{6.3}$$

The height of the front face, h_f, is then calculated by dividing the width of the front face by the aspect ratio of the image, R_a.

$$h_f = \frac{w_f}{R_a} \tag{6.4}$$

Once we have the width and height of the front face, calculating the area of the front face, a_f, is straightforward.

$$a_f = w_f \cdot h_f \tag{6.5}$$

We are interested in the area for each pixel of the image, or other words the front cell area, a_c.

Front cell area, a_c The front cell area is the area for each pixel in the image for the front face area, a_f, at each distance, d_f.

The front cell area, a_c, is calculated using the front face area, a_f, the pixel width, w_p, and the pixel height, h_p. The cell areas are in general bigger or much bigger than a snowflake area.

$$a_c = \frac{a_f}{w_p \cdot h_p} \tag{6.6}$$

The figure below shows how the front face area is divided into the front cell areas.

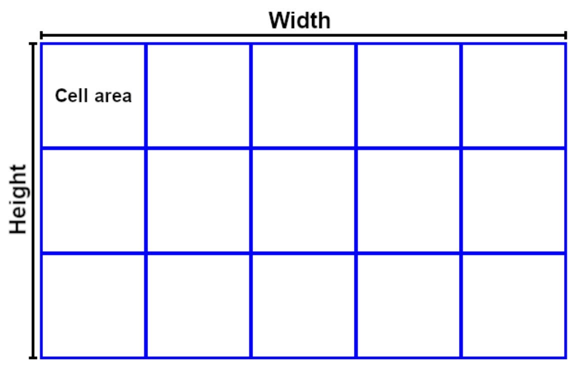

Figure 6.3.: This figure shows how an area is subdivided into the cell areas for an image of size 5x3 pixels.

If we consider a snowflake to be a sphere with a radius, r_s, of 2.5 mm, we can calculate the maximum number of snowflakes that fit into a cell area. The number of snowflakes is calculated by dividing the cell area by the area of a snowflake, a_s.

Area of a snowflake, a_s The area of a snowflake is the area of a circle with a radius of a snowflake, r_s.

$$a_s = \pi \cdot r_s^2 \tag{6.7}$$

The table below shows the cell area and several snowflakes calculated at different distances for an image of size 1920x1080 and a horizontal opening angle of 110 degrees.

Distance (m)	**Cell area** (mm^2)	**# of snowflakes**
1	2.213	0.113
2	8.852	0.451
2.979	19.64	1.0
5	55.328	2.818
10	221.311	11.271
20	885.246	45.085
50	5532.787	281.783
100	22131.149	1127.13
200	88524.598	4508.521
500	553278.735	28178.255
1000	2213114.94	112713.018

Table 6.2.: This table shows the front area (mm^2) of a single cell at different distances (m) for an image of size 1920x1080 pixels, $R_a = \frac{16}{9}$, with a horizontal viewing angle, $\theta_w = 110$. It also shows how many snowflakes would fit in that area if a snowflake is considered as a sphere with a radius, r_s, of 2.5 mm.

Minimum distance, d_{min} The minimum distance is the distance at which the area of the snowflake, a_s, equals the front cell area, a_c. It is the closest distance at which we start slicing the pyramid view.

To find the equation for the minimum distance, d_{min}, we start with the equation for the area of the cell, a_c. Then we expand it using the equations above to get an equation with the area of the cell and the distance, d_f, in it.

$$a_c = \frac{w_f \cdot h_f}{w_p \cdot h_p} \tag{6.8}$$

$$a_c = \frac{w_f \cdot \frac{w_f}{R_a}}{w_p \cdot h_p} \tag{6.9}$$

$$a_c = \frac{w_f^2 \cdot \frac{h_p}{w_p}}{w_p \cdot h_p} \tag{6.10}$$

$$a_c = \frac{w_f^2}{w_p^2} \tag{6.11}$$

$$a_c = \frac{(\tan\left(\frac{\theta_w}{2}\right) \cdot 2 \cdot d_f)^2}{w_p^2} \tag{6.12}$$

$$a_c = \frac{\tan^2\left(\frac{\theta_w}{2}\right) \cdot 4 \cdot d_f^2}{w_p^2} \tag{6.13}$$

$$\tag{6.14}$$

Once we have this equation we replace the area of the cell, a_c, with the area of a snowflake, a_s. We then solve for the distance to get the distance at which the area of the cell equals the area of a snowflake. In other words, this would be the minimum distance, d_{min}.

$$d_{min} = \sqrt{\frac{a_s \cdot w_p^2}{4 \cdot \tan^2\left(\frac{\theta_w}{2}\right)}} \tag{6.15}$$

We can now calculate the minimum distance, d_{min}, for an image with pixel width, w_p, and a horizontal opening angle, θ_w. In the table above you can see the minimum distance where the number of snowflakes in the cell area equals 1.

6.1.3. Calculating the volume of the cell

We want to look at the volume between the front cell area, a_c, and the back cell area, a_{c+1}.

Back cell area, a_{c+1} The back cell area is the area for each pixel in the image for the back face area, a_{f+1}, at the next distance, d_{f+1}.

In the figure below you can see the cell volume, v_c, highlighted in grey from a top-down view.

Cell volume, v_c The cell volume is the volume between the front cell area, a_c, and the back cell area, a_{c+1}.

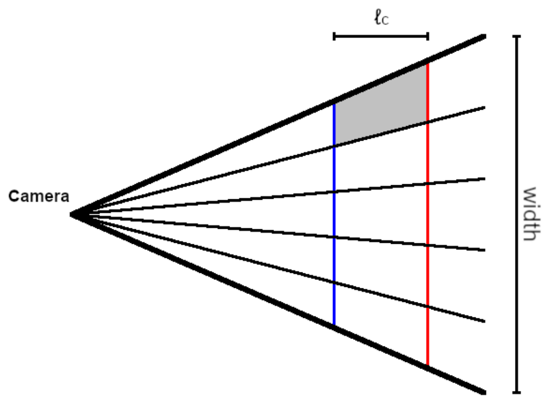

Figure 6.4.: This is a top-down view of the pyramid projection. This is an image with a size of 5x3 pixels. The blue line represents the front face area of the volume cells and the red line represents the back face area of the volume cells. The front and back face areas are placed a cell length apart, ℓ_c and the greyed out area indicates a volume cell, v_c, looked at from the top.

The front and back face areas are then subdivided by the number of pixels in the image to get the cell area as shown in the figure 6.3 for an image of size 5x3 pixels. Another view of the areas divided into cells is shown in the figure 6.4 where you can see a top-down view of this projection.

A single-volume cell is visualized from top-down in the figure6.4. The volume of the cells is calculated by first dividing the front face area by the number of pixels, which results in a grid as shown in the figure6.3. The cell area, a_c, is then calculated in the equation6.6. This is repeated to calculate the back cell area, a_{c+1}.

Then the volume of a single volume cell, v_c, is estimated by taking the average of the front cell area, a_c, and back cell area, a_{c+1}, and multiplying it by the cell length, ℓ_c:

$$v_c = \frac{a_c + a_{c+1}}{2} \cdot \ell_c \tag{6.16}$$

6.1.4. Determining the number of snowflakes in each cell area

We calculate the number of snowflake areas that fit into the cell area at each distance, n_{sca}.

Number of snowflake column areas, n_{sca} The number of snowflake column areas is the number of snowflake areas, a_s, that fit into a cell area, a_c.

Snow column, v_{sc} The volume behind a cell area extending over the distance l_c gives us a snowflake column.

We don't need to know if there are multiple snowflakes in the same snowflake column, or other words if snowflakes are overlapping in the camera view, we don't want to count all the overlapping snowflakes. Instead, we would like to know if there is at least 1 snowflake within the snowflake column behind each subdivided cell area and count that as 1 snowflake.

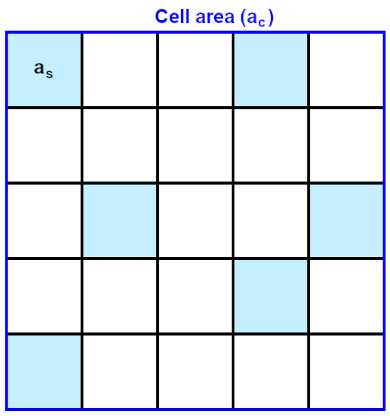

Figure 6.5.: This grid represents a single cell area. It is subdivided into sections equal to the area of a snowflake. It should be noted that the area of a snowflake is shown as a square, but we calculate it as the area of a circle with a radius, $r_s = 2.5$ mm. The subdivided cells represent the column of snow volumes from the front to the back face. The highlighted cells is showing that there is at least 1 snowflake in that snow column.

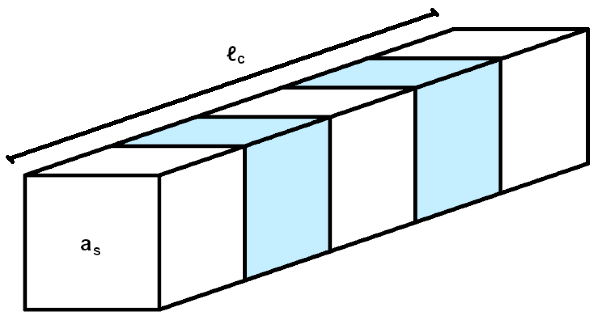

Figure 6.6.: This figure shows a snow column and how it could potentially contain snowflakes. Each cube in the series is equal to the volume of a snowflake. It should be noted that we calculate the volume as the volume of a sphere and not a cube. The highlighted cubes contain a snowflake.

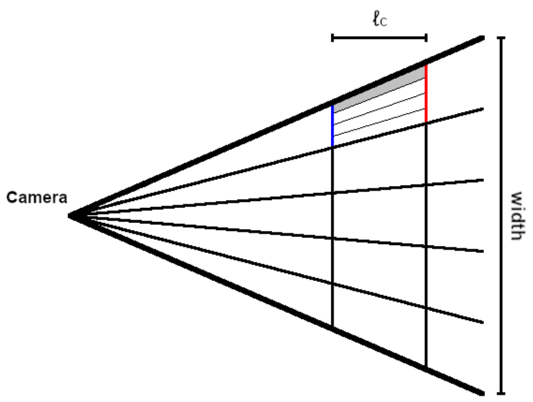

Figure 6.7.: The figure shows the top-down view of a pyramid projection of an image with pixel width, $w_p = 5$ pixels. A single-volume cell is subdivided by seeing how many snowflakes fit in the front cell area and extending that to the back cell area. The front and back cell area are represented in blue and red respectively. The snowflake column, v_{sc}, behind a subdivided cell area is represented in grey.

Therefore, to determine how many snowflakes are in each cell area, a_c, we need to know the probability of at least 1 snowflake, $P_{a\ell}$, in the snowflake column for the subdivided cell area. We would also need to know the number of snowflake areas, n_{sca}, that fit into each front cell area, a_c. These variables are used with a binomial random generator to randomly distribute snow in the different cells.

Snow density, D_s The snow density, D_s, is the number of snowflakes desired in a cubic meter.

Volume of a snowflake, v_s The volume of a snowflake is the volume of a sphere with radius, r_s.

$$v_s = \frac{4}{3} \cdot \pi \cdot r_s^3 \tag{6.17}$$

Probability of snow in a snowflake volume The probability of a snowflake in a single snowflake volume is calculated with the snow density, D_s, and the volume of the snow sphere, v_s.

$$P_s = D_s \cdot v_s \tag{6.18}$$

We then calculate the number of snowflake areas that fit into each front cell area or in other words the number of subdivided cell areas, n_{sca}. An example of this grid is shown in the figure 6.5.

$$n_{sca} = \frac{a_c}{a_s} \tag{6.19}$$

The number of subdivided cell areas, n_{sca}, is then used to calculate the snowflake column volume, v_{sc}, behind each subdivided cell area. This snowflake column, v_{sc}, is visualized in the figure 6.6.

$$v_{sc} = \frac{v_c}{n_{sca}} \tag{6.20}$$

Furthermore, we need to calculate the number of snowflakes, v_s, that fit into the snowflake column, $v_s c$. This number is represented as n_{scv}.

$$n_{scv} = \frac{v_{sc}}{v_s} \tag{6.21}$$

We also need to know the probability of no snowflake in a snowflake volume, P_{ns}:

$$P_{ns} = 1 - P_s \tag{6.22}$$

After that we can finally calculate the probability of at least 1 snowflake for a subdivided cell area:

$$P_{a\ell} = 1 - (P_{ns})^{n_{scv}} \tag{6.23}$$

Now with the number of subdivided cell areas, n_{sca}, and the probability of at least 1 snowflake in a subdivided cell area, $P_{a\ell}$, we can use these variables in the binomial random generator to generate snowflakes for that cell area.

$$A_s = binom(n_{sca}, P_{a\ell}) \tag{6.24}$$

We want to know how many successes (snowflakes) we have in the cell area. To find that we try to generate at least 1 snowflake for each of the subdivided cell areas. This gives us the number of snowflakes for that cell area, A_s. In the graphs below the amount of snowflakes against the number of cells is plotted.

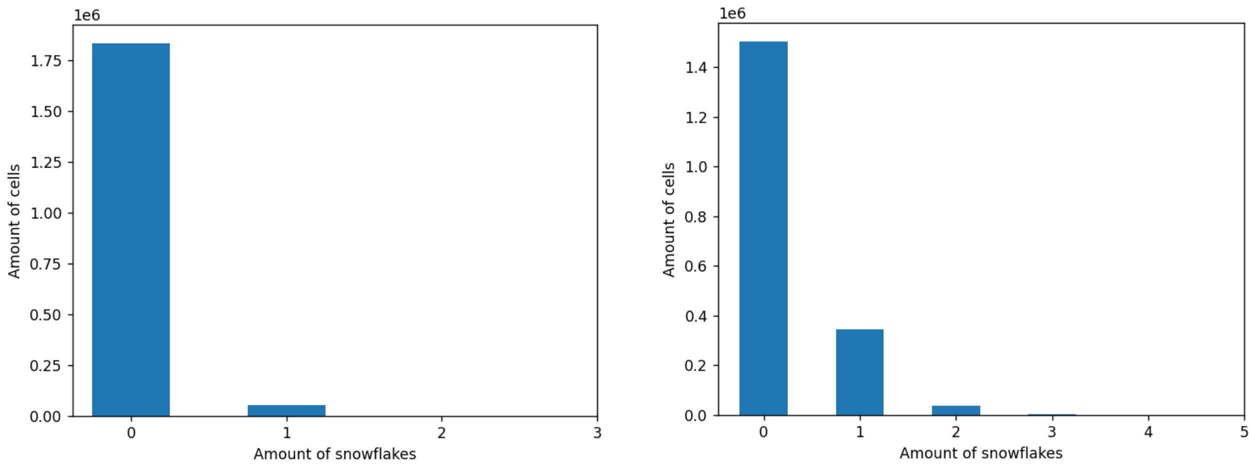

Figure 6.8.: On the left you can see the distribution of snowflakes at a distance of 30 meters from the camera. On the right, you can see the distribution of snowflakes at a distance of 50 meters from the camera. Both graphs is taken from an image of size 1920x1080, a horizontal viewing angle of $\theta_w = 110$ degrees and a snow density, $D_s = 10^4$.

6.1.5. Calculating the new brightness of the pixels

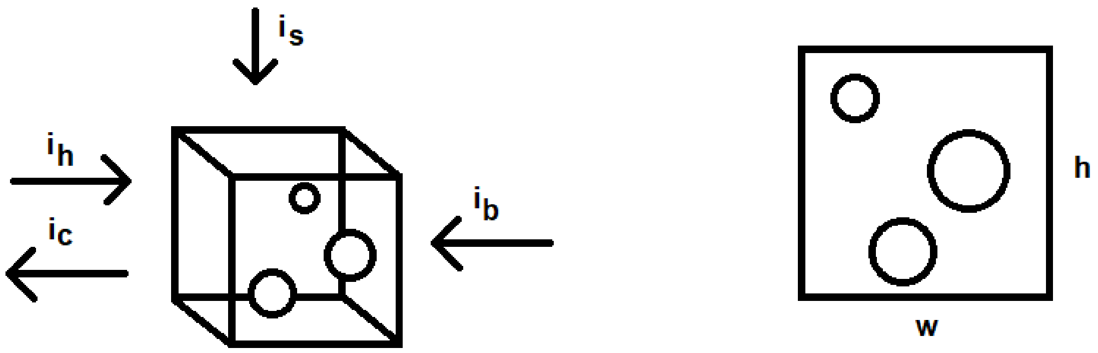

Figure 6.9.: On the left you can see a volume cell represented as a cube. The i_b is the light from the background, i_s is the snow brightness and i_c is the light sent to the camera. The variable i_h is not used yet, but it is the light from the camera which might be reflected. On the right, it is shown how the front face of this volume would look to the camera.

Once we know the amount of snowflakes in each cell area we can calculate how much the snow brightens the corresponding pixel. We calculate this by first calculating the ratio of snow in the cell, R_s. This is calculated with the area of a single snowflake, a_s, the number of snowflakes in that cell, A_s, and the front cell area, a_{fc}. Note that we assume that none of the snowflakes overlap.

$$R_s = \frac{a_s * A_s}{a_c} \tag{6.25}$$

If there are no snowflakes, $A_s = 0$, in the cell then the ratio of snow is, $R_s = 0$, and that cell does not affect the pixel it corresponds to.

To calculate the pixel values for the current pixel, p_d, the ratio of snow, R_s, and the snow brightness, i_s, is used to augment the previous pixels values, p_{d+1}. The pixel values start to be calculated from the furthest distance (d_{max}) to the closest distance (d_{min}). This means that the pixel values are first calculated at the maximum distance.

$$p_d \quad = R_s \cdot i_s + (1 - R_s) \cdot p_{d+1} \tag{6.26}$$

6.2. Implementation

The implementation of the snow model takes the parameters: image, θ_w, d_{min}, d_{max}, i_s and D_s.

The image is the original image before it is converted into an array. The θ_w parameter is the horizontal opening angle of the camera. The maximum distance, d_{max}, is the furthest distance to project the pyramid view to. The snow brightness, i_s, indicates the brightness of the snow, a higher value the brighter the snow. The snow density, D_s, indicates how much snow to simulate on the image, higher value is equal to more snow. Finally, the minimum distance, d_{min}, is the closest distance to calculate a front face at.

In the implementation of the snow model the minimum distance is multiplied by the square root of 2 until it passes the maximum distance. All the resulting values are the distance which the pyramid view is slices at.

The source code for this implementation of the snow model is given in appendix B.

6.3. Challenge with the snow model

The snow model has a bug that was discovered late in the thesis, and a solution is yet to be reached. It involves that the density of the snow in the augmented image does not match what we would expect to see from it.

The expected behavior of the snow model is that when you simulate snow over a set distance of for example 20 meters. It remains the same amount of snow even if it is simulated from 5 to 25 meters and 400 to 420 meters. The only difference would be that the graininess at the closer distance should be large while at the far distance it should be almost no graininess.

To demonstrate this bug we simulated snow from 5 to 25 meters to get this:

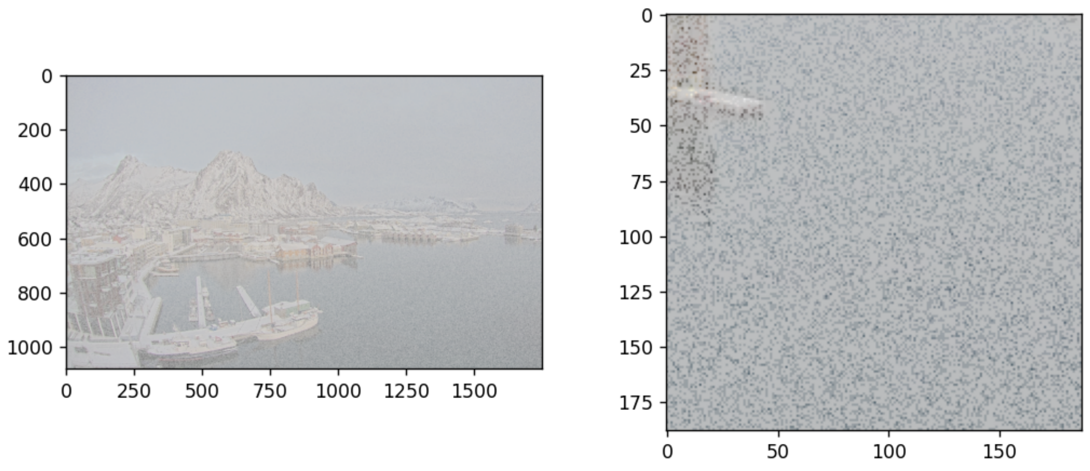

Figure 6.10.: Here we see the expected graniness from the close distances

Once we move the 20 meter volume to a far distance from 400 to 420 meters we get:

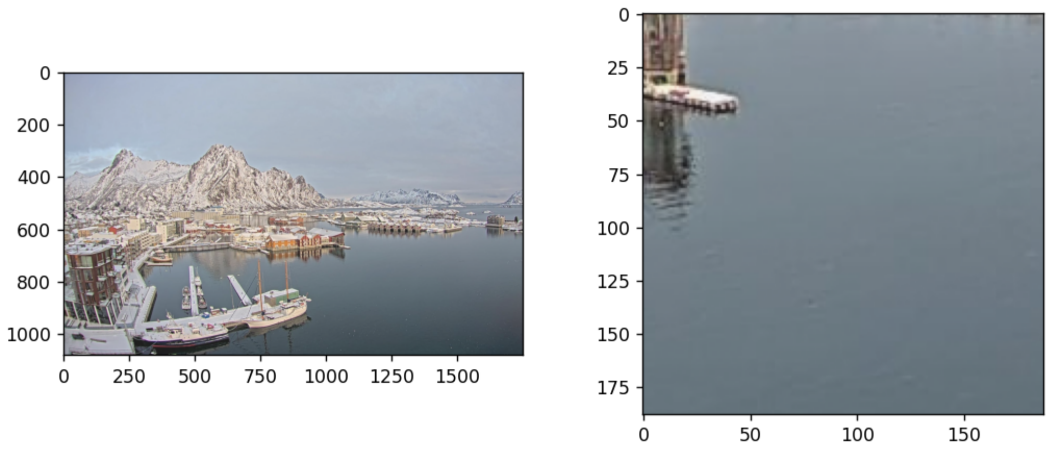

Figure 6.11.: Here we can see that there is almost no snow even though we would expect the amount of snow to remain the same.

7. Results

In this thesis, we have experimented with ways to simulate weather on single images in maritime environments. This has been done to look at the possibilities of expanding datasets with synthetic images of weather. Towards this goal, we have looked at some datasets with maritime scenes that already exist and what types of weather images they contain. We have collected our dataset of images exclusively from maritime scenes. Next, we've looked at a few different depth estimation architectures to see how they performed on the newly created maritime dataset. Lastly, we presented two weather models, for fog and snow, that can simulate weather on single images using depth information.

Here we will present the results of our work.

7.1. Dataset

We collected a new dataset for this thesis from web cameras around Norway. It was important for us to collect this dataset to get a sufficient amount of maritime scenes to estimate depth on and simulate weather on. At the same time it gave us the opportunity to capture different weather conditions in those same scenes.

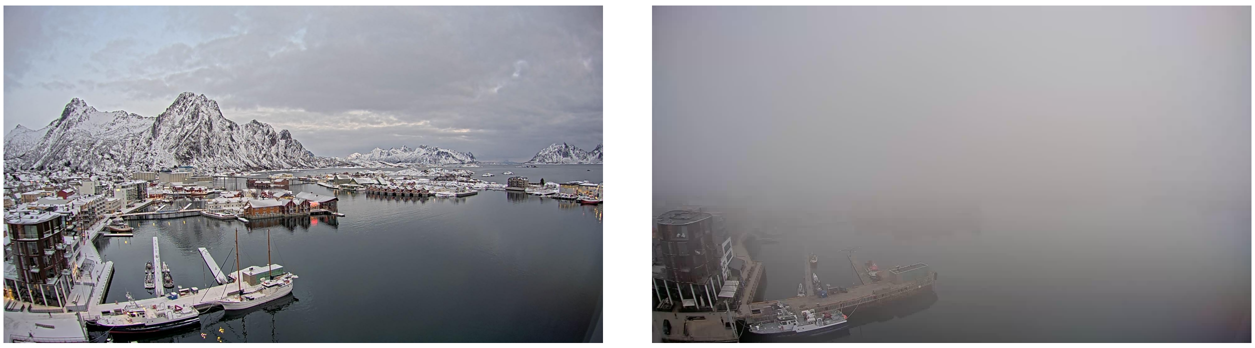
Figure 7.1.: Examples of images from the dataset. This is the Svolvær scene.

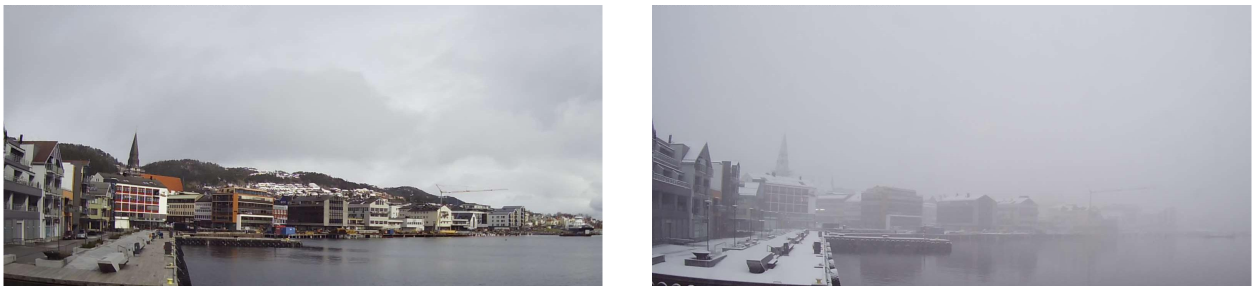
Figure 7.2.: Examples of images from the dataset. This is the Molde Harbor scene.

Over the period of a few months we collected over 3000 images from several different web cameras. This gave us a lot of different types of lighting and weather conditions to choose from. Even though we did not need that amount of images in this thesis, we can see that a dataset like this collected over an even longer period of time could prove highly useful in future computer vision projects involving maritime scenes.

7.2. Depth estimation

In this thesis we wanted to see how the three depth estimation architectures performed on maritime scenes. The results were varied and most of the depth maps produced by any of the depth estimators did not prove useful. MiDaS 3 was very consistent between different lighting and scenes, but it was always worse than the best depth predicted by 3D Ken Burns with correct lighting. 3D Ken Burns was very inconsistent with its depth estimation, sometimes it produced pretty accurate results while other times producing worse results than MiDaS 3.

The most accurate depth estimation on some images were given by 3D Ken Burns, and below you can see a couple of the best depth estimation from it.

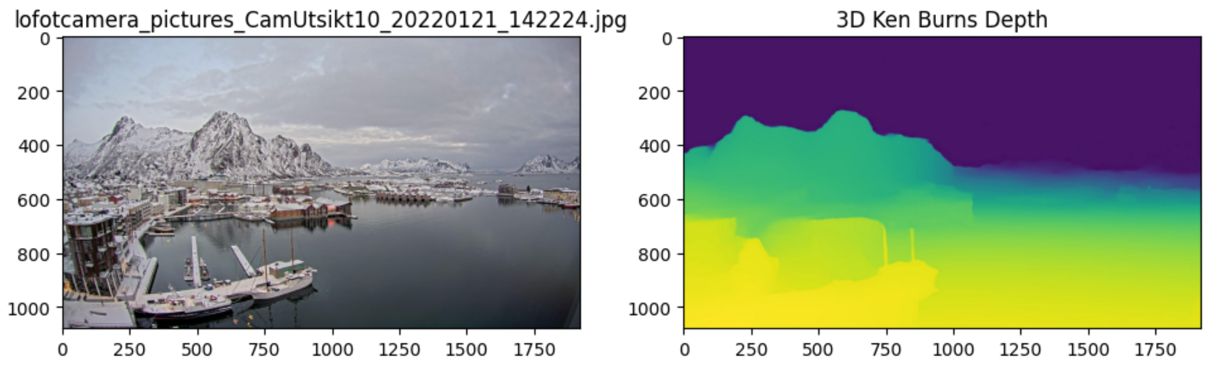

Figure 7.3.: On the left you see the original image from the Svolvær scene. On the right is the depth estimated with 3D Ken Burns.

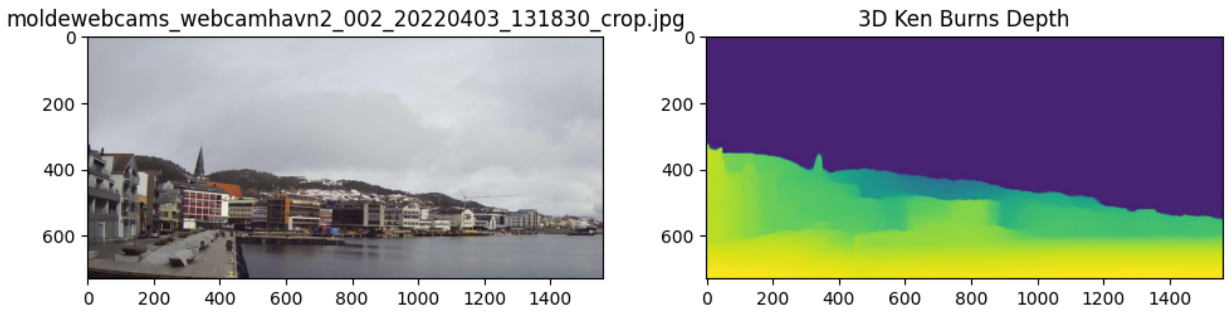

Figure 7.4.: On the left you see the original image from the Molde Harbor scene. On the right is the depth estimated with 3D Ken Burns.

7.3. Fog model

The fog model is simple. It can only produce homogeneous fog and adjusting the colors of the fog is not easily done. Even with these limitations, it can still produce some fairly convincing results as is shown below.

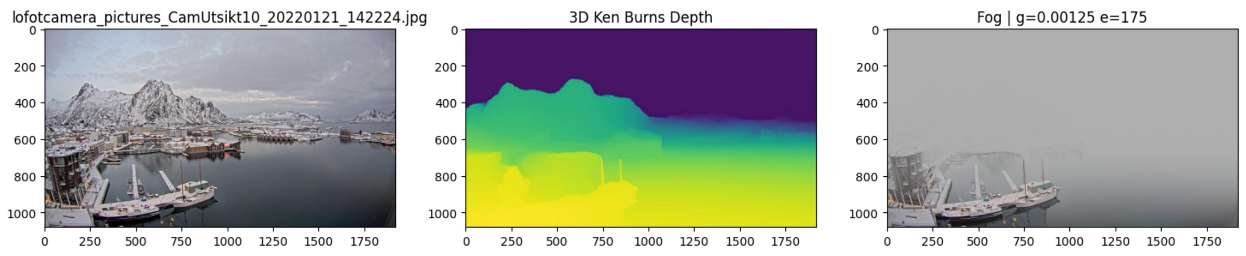

Figure 7.5.: On the left you see the original image. In the middle is the depth estimation from the 3D Ken Burns architecture. On the right is the original image augmented with fog with gamma, g=0.00125, and environmental light, e= 175.

There are two variables that one can change to adjust the type of fog that is going to be produced. The gamma value, γ, is adjusting the amount of fog to add to the image. A higher gamma value results in a higher fog density.

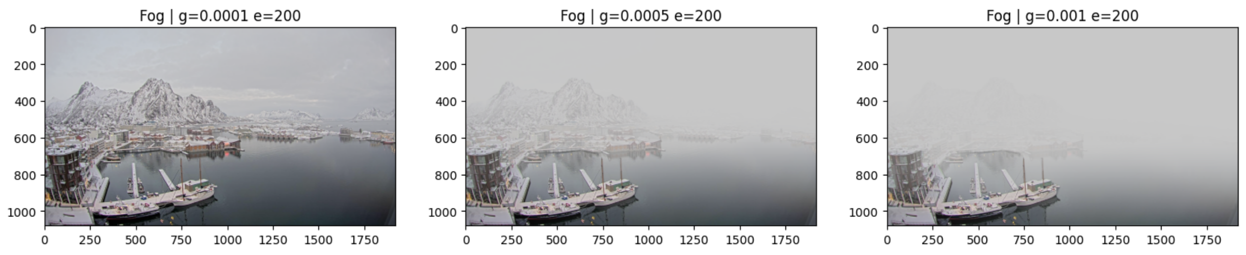

Figure 7.6.: These are examples of fog generated with different values of gamma, g. The values for each image are noted above the images. As shown in the images, a higher value of gamma gives a higher fog density. The original image and depth estimation is the same as for figure 7.5.

The environmental light variable, e, adjusts the amount of light to add the fog. The higher value for environmental light the lighter the fog. A value of 0 means completely dark fog while a value of 255 should be completely white fog.

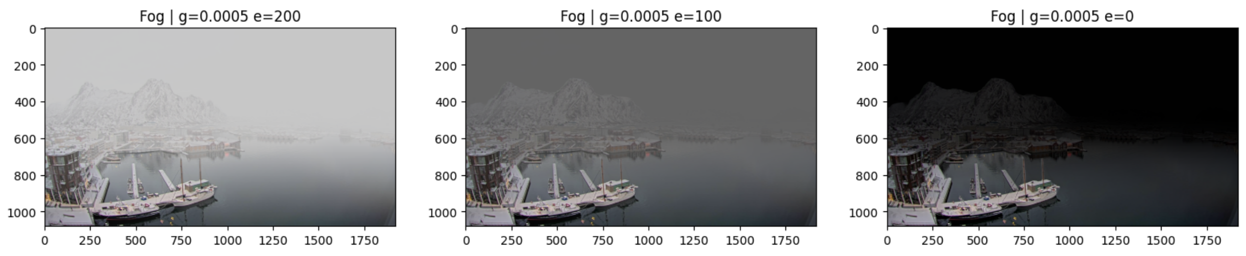

Figure 7.7.: These are examples of fog generated with different values of environmental light, e. The values for each image is noted above the images. As shown in the images, it is possible to adjust the brightness of the fog with a higher or lower value for the environmental light. The original image and depth estimation is the same as for figure 7.5.

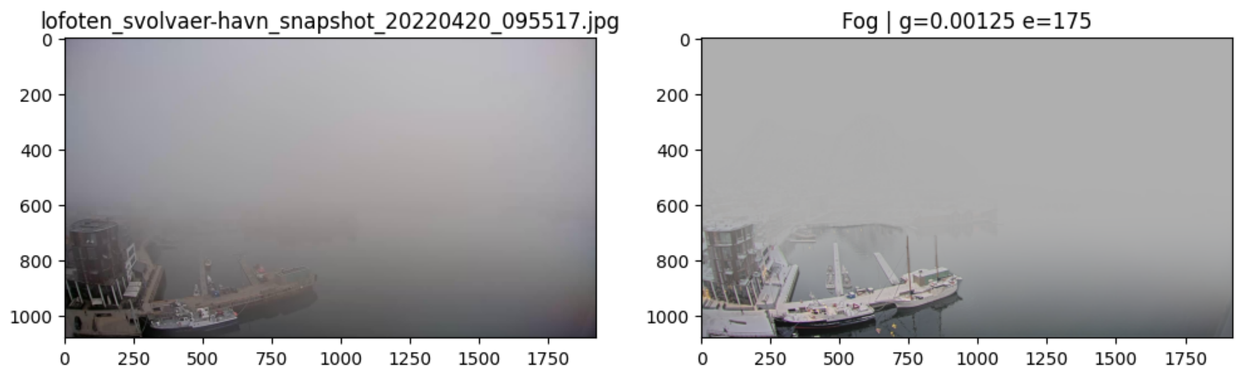

Figure 7.8.: The image on the left is an image of real fog from the Svolvær scene. The image on the right is augmenting the original image shown in the figure 7.5 with fog trying to match the real fog image. The values for the augmented fog is shown above the image.

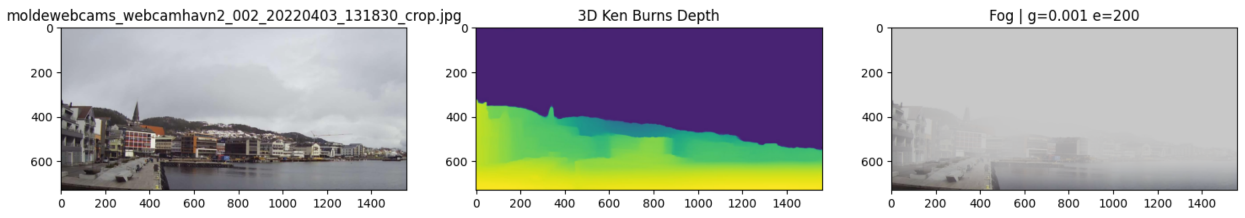

Figure 7.9.: On the left you see the original image. In the middle is the depth estimation from the 3D Ken Burns architecture. On the right is the original image augmented with fog with gamma, g=0.001, and environmental light, e= 200.

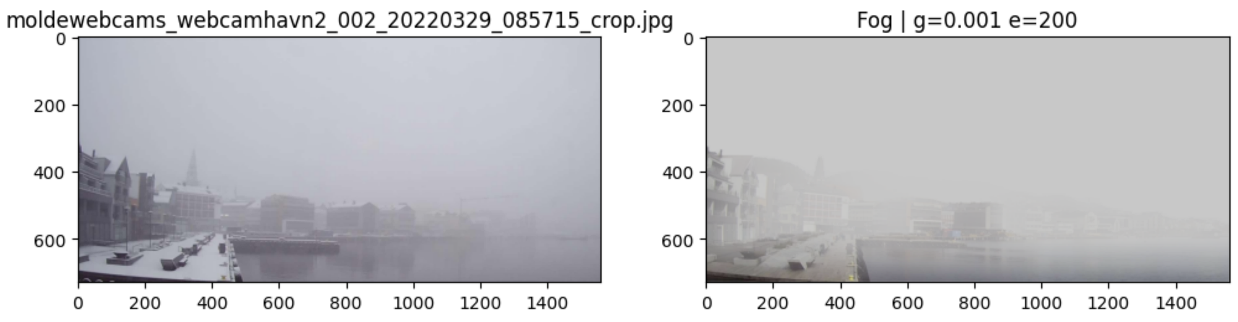

Figure 7.10.: The image on the left is an image of real fog from the Molde Havn scene. The image on the right is augmenting the original image shown in figure 7.9 with fog trying to match the real fog image. The values for the augmented fog is shown above the image.

7.4. Snow model

The snow model still needs to be improved. There are challenges to be solved and improvements to be made. Here we will show briefly what type of snow it can simulate as it is. The amount of snow added to an image is chosen with the density variable, D_s, where a higher value means more snow. In the same way that you can alter the brightness of the fog in the fog model with the environmental light variable, you can also alter the brightness of the snow with the snow brightness variable, i_s.

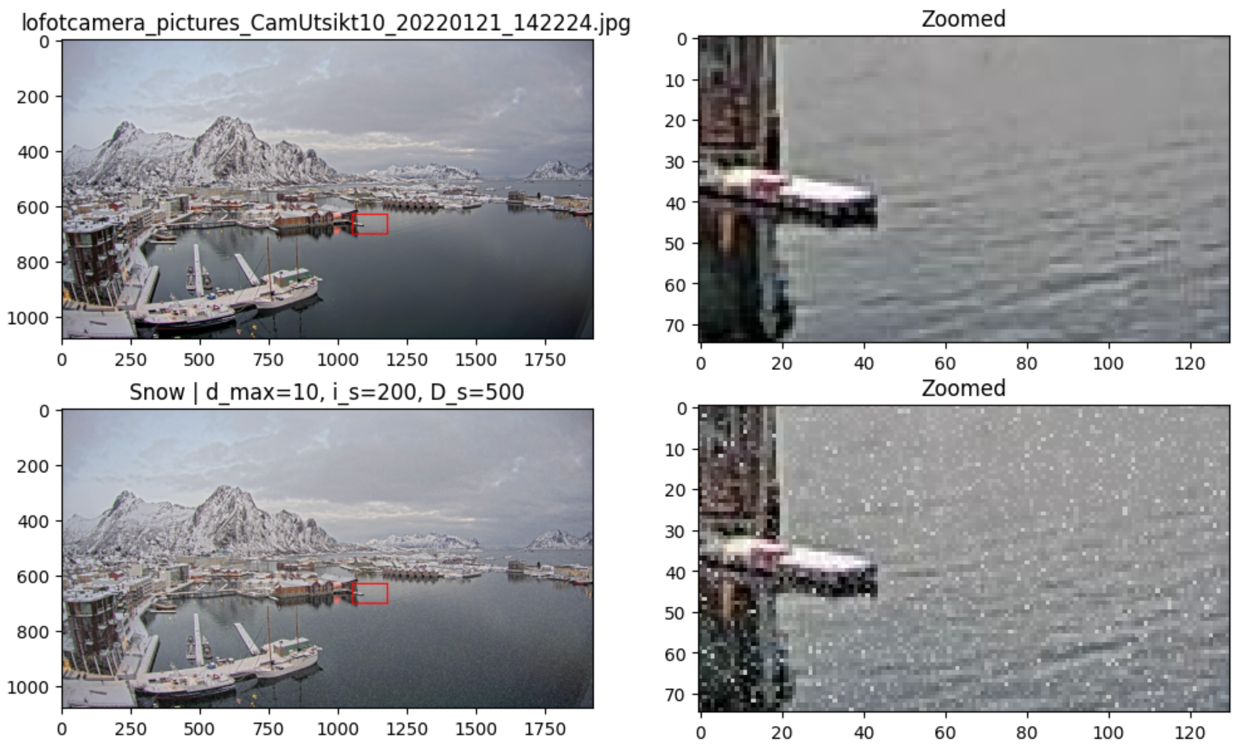
lofotcamera_pictures_CamUtsikt10_20220121_142224.jpg
Zoomed
Snow | d_max=10, i_s=200, D_s=500
Zoomed

Figure 7.11.: The image in the top left is the original image from the Svolvær scene. The image in the bottom left is an image augmented with snow. The parameters for the snow model is noted above the image. The images on the right are zoomed-in versions of the left images.

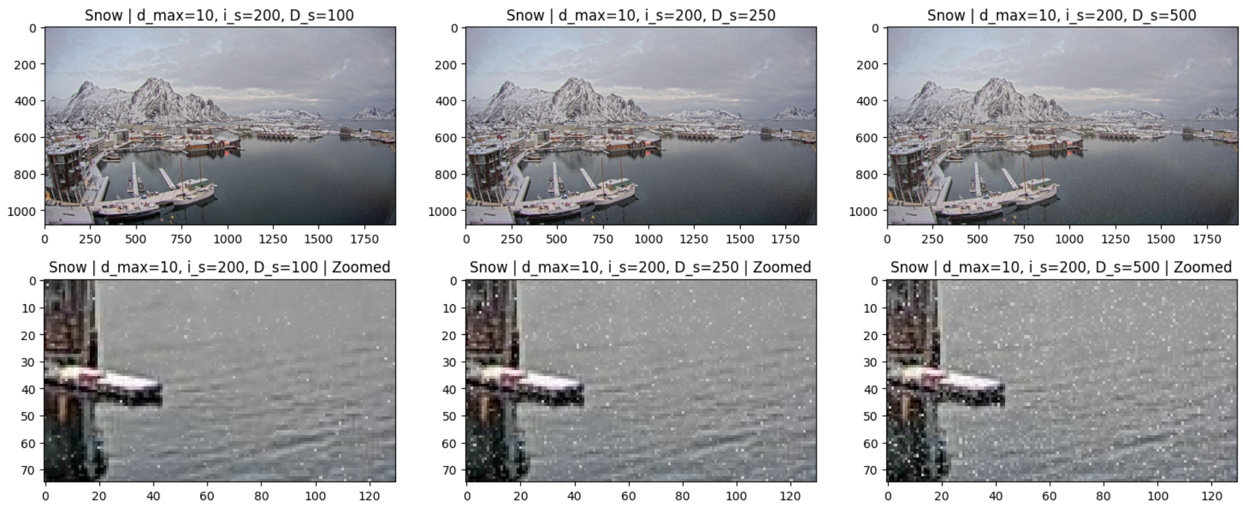
Snow | d_max=10, i_s=200, D_s=100
Snow | d_max=10, i_s=200, D_s=250
Snow | d_max=10, i_s=200, D_s=500
Snow | d_max=10, i_s=200, D_s=100 | Zoomed
Snow | d_max=10, i_s=200, D_s=250 | Zoomed
Snow | d_max=10, i_s=200, D_s=500 | Zoomed

Figure 7.12.: This figure shows how the snow density variable, D_s, changes the amount of snow in the augmented image. The number of snow increases from left to right, see the parameters at the top of the image. The images at the bottom are just zoomed-in versions of the images at the top.

8. Conclusion

During this thesis we have collected a new dataset of maritime images that includes different types of weather. This dataset proved useful in this thesis, and we also believe that the same or similar dataset collected in the future can prove to be very useful in similar future projects.

We have also investigated how some depth estimation architectures perform on maritime images. The results from this was varied as the quality of the depth estimation was strongly dependent on the lighting and others factors in the image used. The depth estimators tested in this thesis did provide some good results. However, monocular depth estimation in maritime scenes is not stable as it stands now and needs to be improved in the future if this is going to be a viable strategy to estimate depth in maritime images.

The fog model presented in this thesis is very simple. It only produces homogeneous fog and is highly dependent on the quality of the depth information it is given. There are several improvements that can be made in the future. Some of the improvements that can be made are; heterogeneous fog density, adjusting the color of the fog and adjusting the fog density by elevation.

The snow model we have presented in this thesis still has some challenges to overcome. While it can produce snow smaller than the size of a pixel it cannot simulate snow larger than that yet. This is an important step to simulate real snow and should looked into as an improvement in the future. Another thing is that at a certain distance away from the camera, snow begins to look like fog. Therefore, it should be possible to use the fog model to simulate the snow after that point. The snow model is significantly slower than the fog model and by doing this we only need to simulate snowflakes up close to the camera. This will lower the time it takes to augment an image with snow.

Bibliography

Google (2022). *Overview of Svolvær in Google Maps.* URL: https://www.google.com/maps/.

Halder, Shirsendu Sukanta, Jean-François Lalonde, and Raoul de Charette (2019). “Physics-Based Rendering for Improving Robustness to Rain”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV).*

Lin, Tsung-Yi et al. (2014). *Microsoft COCO: Common Objects in Context.* DOI: 10.48550/ARXIV.1405.0312. URL: https://arxiv.org/abs/1405.0312.

Niklaus, Simon et al. (2019a). “3D Ken Burns Effect from a Single Image”. In: *ACM Transactions on Graphics* 38.6, 184:1–184:15.

– (2019b). *GitHub Repository for 3D Ken Burns.* URL: https://github.com/sniklaus/3d-ken-burns.

NumPy (2022). *NumPy’s Polyfit function.* URL: https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html.

Ranftl, René, Alexey Bochkovskiy, and Vladlen Koltun (2021). *Vision Transformers for Dense Prediction.* arXiv: 2103.13413 [cs.CV].

Ranftl, René et al. (2020). *Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer.* arXiv: 1907.01341 [cs.CV].

Rondestvedt, Trond Jacob (2021). *Unpublished NTNU pre-project report.*

Sen, Prithwish, Anindita Das, and Nilkanta Sahu (2021). “Rendering Scenes for Simulating Adverse Weather Conditions”. In: *Advances in Computational Intelligence.* Ed. by Ignacio Rojas, Gonzalo Joya, and Andreu Català. Cham: Springer International Publishing, pp. 347–358. ISBN: 978-3-030-85030-2.

Shao, Zhenfeng et al. (Aug. 2018). “SeaShips: A Large-Scale Precisely-Annotated Dataset for Ship Detection”. In: *IEEE Transactions on Multimedia* 20, pp. 1–1. DOI: 10.1109/TMM.2018.2865686.

Von Bernuth, Alexander, Georg Volk, and Oliver Bringmann (2019). “Simulating photo-realistic snow and fog on existing images for enhanced CNN training and evaluation”. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC).* IEEE, pp. 41–46.

Zhang, Ning, Lin Zhang, and Zaixi Cheng (Oct. 2017). “Towards Simulating Foggy and Hazy Images and Evaluating Their Authenticity”. In: pp. 405–415. ISBN: 978-3-319-70089-2. DOI: 10.1007/978-3-319-70090-8_42.

A. Source Code - Fog model

```python
# Python code for the fog model
# Parameters are set at the bottom
# Outputs image augmented with fog
import os
import sys
import time
import numpy as np
from PIL import Image

# im_filepth      -> Filepath to the image to augment
# depth_filepth   -> Filepath to the depth values (.npy), expects a depth value for
    each pixel.
# output_filepth -> Where do you want the augmented image to be outputted.
# gamma           -> Determines how much the original light from the scene is
   reduced.
# e               -> Determines the amount of environmental light to add to the
   scene.
# delta           -> length of a fog volume
def fog_model(image, depth_array, output_filepth, gamma, e, delta):
    image_array = np.array(image)
    # The depth values are not predicted close to real distance so we need to
   normalize the values to close to real values
    # We assume the sky is 10 km away and the closest depth is at 0 meters.
    depth_array = np.interp(depth_array, (depth_array.min(), depth_array.max()),
   (10000, 0))


    # Needed to make sure the dimensions are correct.
    if depth_array.shape[0] != image_array.shape[0] or depth_array.shape[1] !=
   image_array.shape[1]:
        depth_array = np.rot90(depth_array)

    # Converting gamma to alpha
    alpha = 1 - gamma

    # Creating a temporary image array with dtype=uint16
    # The reason is that when adding up the values in the equation,
    # they can overflow and cause glitches in the final augmented image.
    aug_image_uint16 = np.empty(shape=image_array.shape, dtype=np.float16)

    # If the image has a 4th channel for opacity (RGBA image) then set that value
   to 255 (we don't want any opacity).
    if(image_array.shape[2]==4):
        aug_image_uint16[:,:,3] = 255


    # First part of the equation
    part1 = np.power(alpha, (depth_array[:,:]/delta))

    # Iterating through the RGB values for each pixel
    aug_image_uint16[:,:,0] = (part1[:,:] * image_array[:,:,0] + (1 - alpha) * e *
    (part1[:,:]-1)/(alpha-1))
    aug_image_uint16[:,:,1] = (part1[:,:] * image_array[:,:,1] + (1 - alpha) * e * (part1[:,:]-1)/(alpha-1))
    aug_image_uint16[:,:,2] = (part1[:,:] * image_array[:,:,2] + (1 - alpha) * e * (part1[:,:]-1)/(alpha-1))


    # Cutting down any values above 255 down to 255 to avoid overflow
    np.putmask(aug_image_uint16, aug_image_uint16>255, 255)

    # Converting back to uint8
    aug_image = np.empty(shape=image_array.shape, dtype=np.uint8)
    aug_image[:,:,:] = aug_image_uint16[:,:,:]

    return aug_image


if __name__ == "__main__":
    im_filepth = "./report/molde_havn/moldewebcams_webcamhavn2_002_20220403_131830_crop.jpg"
    depth_filepth = "."+im_filepth.split(".")[1]+"_3d-ken-burns-depth.npy"
    output_filepth = "./report/molde_havn/"

    # Loading the image into a numpy array
    im_file = im_filepth.split("/")[-1]
    im_name = im_file.split(".")[0]
    image = Image.open(im_filepth)

    # The depth map is stored as a numpy array in a .npy file
    # Loading the depth array
    depth_array = np.load(depth_filepth)

    g = 0.0008
    e = 190
    d = 1

    aug_image = fog_model(image,depth_array,output_filepth,g,e,d)

    # Converting the array to an image
    aug_im = Image.fromarray(aug_image)

    # Saving the image
    depth_method = depth_filepth.split("/")[-1].split("_")[-1].split(".")[0]
    save_name = output_filepth+im_name+"_g"+str(g).split(".")[0]+"+"+str(g).split(".")[1]+"_e"+str(int(round(e,1)))+"_"+depth_method+".png"
    aug_im.save(save_name)
    aug_im.show()
```

B. Source Code - Snow model

```python
# Python code for the snow model
# Parameters are set at the bottom
# Outputs image augmented with snow
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

def calculate_cell_area (d, w_p, h_p,theta_w):
    # Front face width and height
    w_f = np.tan(theta_w/2) * 2 * d
    h_f = w_f / (w_p/h_p)

    # Front face area
    a_f = w_f * h_f

    # Front cell area
    a_c = a_f / (w_p * h_p)

    return a_c

def snow_model_abbrv (image, theta_w, d_min, d_max, i_s, D_s):
    image_array = np.array(image)
    aug_image_array = np.full(image_array.shape, image_array, dtype=float)
    w_p = image.width
    h_p = image.height

    # Snowflake Sphere
    r_s = 0.0025 # 0.0025 m -> 2.5 mm in radius
    a_s = np.pi * (np.power(r_s, 2)) # area of a snowflake (circle)
    v_s = (4/3) * np.pi * np.power(r_s, 3) # volume of a snowflake (sphere)

    # Determining the distances to split the pyramid at
    # Distance array
    d_array = []
    # Starts at the closest distance, but this array get reversed
    d_array.append(d_min)

    # Multiplies the previous value in the distance array with the square root of
    2
    while d_array[-1] < d_max:
        d_array.append(d_array[-1]*np.sqrt(2))

    # We want to start calculating from the furthest distances so we reverse the
    array before looping through it
    d_array.reverse()

    # Initiating the random generator before the loop (used for binomial
    distribution)
    # NB! I'm using an arbitrary seed for testing purposes, if the algorithm is
    going to generate random snow every time don't use a seed.
    rng = np.random.default_rng(678124)

    # Iterating through the distances.
    for (i, d) in enumerate(d_array):
        # Need to skip the first distance (the furthest away)
        # This is done because we need a back face to calculate a volume to.
        if(i==0):
            continue

        # Front Cell Area
        a_c  = calculate_cell_area(d,w_p,h_p,theta_w)
        d1 = d_array[i-1] # Distance to the back face
        # Back Cell Area
        a_c1  = calculate_cell_area(d1,w_p,h_p,theta_w)

        # Cell length
        l_c = d_array[i-1] - d_array[i]
        # Cell volume, estimated
        v_c = ( (a_c + a_c1) / (2) ) * l_c
        # The probability of a snowflake in a snowflake volume
        P_s = D_s * v_s
        # Number of snowflake areas in cell area
        n_sca = a_c / a_s
        # Volume of the snowflake column behind a cell area
        v_sc = v_c / n_sca
        # Number of snowflake volumes in the snowflake column
        n_scv = v_sc / v_s
        # Probability of no snow in a snowflake area
        P_ns = 1 - P_s
        # Probability of at least 1 snowflake in a snowflake column
        P_al = 1 - np.power(P_ns, n_scv)

        # Generating an amount of snowflakes for each pixel using a binomial random generator
        A_s = rng.binomial(n_sca, P_al, (image.height, image.width))
        # Calculating the ratio of snowflake area to cell area
        R_s = (a_s * A_s) / a_c

        # Altering the brightness of each channel of the pixels using the ratio, R_s
        # Numpy notation, this can be done with regular Python but that is much slower
        aug_image_array[:,:,0] = R_s[:,:] * i_s + (1-R_s[:,:]) * aug_image_array[:,:,0] # Red
        aug_image_array[:,:,1] = R_s[:,:] * i_s + (1-R_s[:,:]) * aug_image_array[:,:,1] # Green
        aug_image_array[:,:,2] = R_s[:,:] * i_s + (1-R_s[:,:]) * aug_image_array[:,:,2] # Blue

    # Returning the augmented image array
    aug_image = np.empty(shape=image_array.shape, dtype=np.uint8)
    aug_image[:,:,:] = aug_image_array[:,:,:]
    return aug_image


if __name__ == "__main__":
    # Path to the image wanting to augment
    image_path = "./input/lofoten/lofotcamera_pictures_CamUtsikt10_20220121_132220.jpg"

    # Parameters for the model
    image = Image.open(image_path)
    # Opening angle of the camera
    theta_w = np.radians(110)
    # The minimum distance from the camera to start calculating snow at
    d_min = 100
    # The cut off distance for snow, not going to be exact because of the way
   distances are calculated.
    d_max = 120
    # Brightness of the snow
    i_s = 200
    # Density of the snow
    D_s = 400

    # Running the model which returns the augmented image
    aug_image = snow_model_abbrv(image, theta_w, d_min, d_max, i_s, D_s)

    # Converting the numpy array back to image format
    aug_im = Image.fromarray(aug_image)
    aug_im.show()

    # Saving the image
    aug_im.save("./output/augmented_image.png")
```