

Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

**Verdeckung von Augmented Reality Windparks in
Großer Entfernung**
**Occlusion for Augmented Reality Windfarms in Great
Distance**

Bachelorarbeit
Informatik

Januar 2023

Vorgelegt von Presented by	Valentino Geuenich Matrikelnummer: 395000 valentino.geuenich@rwth-aachen.de
Erstprüfer First examiner	Prof. Dr. rer. nat. Erika Ábrahám Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University
Zweitprüfer Second examiner	Prof. Dr. rer. nat. Thomas Noll Lehr- und Forschungsgebiet: Software Modellierung und Verifikation RWTH Aachen University
Betreuer Supervisor	Dr. phil. Henning Petzka Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Aachen, im Januar 2023

Valentino Geuenich

Contents

1	Introduction	1
1.1	General Interest	1
1.2	Related Work	1
1.3	Contribution	2
2	Idea Elaboration	2
2.1	Depth Estimation	3
2.1.1	Depth/Structure from Motion (SfM) using SLAM	3
2.1.2	Monocular Depth Estimation (MDE)	5
2.1.3	Digital Elevation Model (DEM) for Depth Estimation	8
2.1.4	Sky Segmentation	11
2.2	Problem Statement The Occlusion Problem	12
2.3	Solution	14
3	Implementation and Insights	17
3.1	Unity Depth from Motion and DEM for Depth Estimation	18
3.1.1	Structure from Motion (SfM) Close-DDE	19
3.1.2	DEM for Depth Estimation Long-DDE/Region Selection	21
3.1.3	Wind Turbine Localisation in the Rendered Image	24
3.1.4	Utility Functionalities	26
3.1.5	Data Extraction	27
3.2	Python Monocular Depth Estimation, Sky Segmentation, Occlusion, and Analysis	28
3.2.1	Data Preparation	28
3.2.2	Monocular Depth Estimation (MDE)	28
3.2.3	Sky-Segmentation-Algorithm	29
3.2.4	Computing Occlusion Maps and Combining Them	30
4	Evaluation	33
4.1	Dataset	33
4.2	Metrics	34
4.3	Evaluation of SfM	35
4.3.1	Fine Tuning SfM Threshold	35
4.3.2	SfM Occlusion with 15m SfM Threshold	37
4.4	Evaluation of Sky Segmentation for Occlusion	38
4.5	Evaluation of SfM + Sky Occlusion	40
4.5.1	Fine Tuning SfM Threshold	40
4.5.2	SfM 30m + Sky Occlusion	41
4.6	Evaluation of MDE	41
4.6.1	MDE vs. MDE + Sky	42
4.6.2	Fine Tuning Region Selection and Thresholding Method for MDE + Sky	43

4.6.3	Evaluation MDE+Sky with Topographic Region Selection and MIN(UF,MAX)	45
4.7	Evaluation of SfM+MDE+Sky	46
5	Conclusion	47
5.1	Limitations	48
5.2	Future Work	50
	References	51
6	Appendix	56
6.1	Graphs	56
6.2	Paulusstraße Dataset	57
6.3	Königshügel Dataset	58
6.4	Informatikzentrum Dataset	59
6.5	Lousberg Dataset	60
6.6	Shader	61

1 Introduction

1.1 General Interest

In the year 2022, the importance of renewable energies took a completely new turn. While renewable energies are the key to mitigating climate change, they have also become essential for Germany's effort to make its energy policy independent of external energy exporters. Next to solar panels, wind farms are one of the essential pillars of the German Energy Revolution (Energiewende). However, according to SWR research, the number of authorizations for new wind turbines is declining for the first half of 2022. The reason for this not only lies in the complicated authorization process but also in the low level of acceptance of wind farms in the population [25].

In order to increase the acceptance in the population, the hybrid system institute pursues a study program for wind farm visualization using augmented reality on smartphones and other handheld devices. If the working group succeeds in visualizing wind farms using augmented reality on smartphones, concerned citizens will be able to get an impression of the visual influence of a planned wind farm in their neighborhood, hoping that this may reassure citizens' concerns and increase the acceptance of wind farms.

However, while the reliable tracking of virtual objects for mobile augmented reality has become a well-solved problem, especially through the use of frameworks like ARCore [17] and ARKit [3], solving the occlusion by deciding what parts of a virtual object should be occluded by the real-world scene, referenced in the following as the occlusion problem, remains an active research area [2, 14, 32, 39, 44].

Solving the occlusion problem is essential for creating an immersive augmented reality experience and will be the topic of this bachelor thesis.

1.2 Related Work

While there has been a lot of work on wind farm visualization in order to increase acceptance among citizens [10, 11, 20, 31], the occlusion problem has widely stayed unaddressed. While Grassi and Klein [20] propose using geographical information to occlude wind turbines, they do not elaborate on this further and do not provide any evaluation of their proposed method.

The lack of solutions for occlusion in great distances may arise from the fact that depth sensors and traditional depth estimation algorithms (stereo and monocular depth estimation) focus on predicting distances of less than 100m for outdoor and 10m for indoor settings [32]. Furthermore, augmented reality on smartphones has limited computational power, and the farther the depth needs to be predicted, the higher the resolution of the processed image must be. In May 2022, Google released a new version of their in-house developed augmented reality framework ARCore, providing 0-65.535m

depth estimation on Android smartphones by processing the smartphone's camera image at a resolution of 120×160 using Depth/Structure from Motion. However, the algorithms' estimation quality significantly decreases after 8m [18], and the already low resolution takes modern smartphones to their computational limits.

1.3 Contribution

It should be mentioned in advance that this bachelor's thesis does not completely solve the occlusion problem for wind farm visualization. As mentioned above, the occlusion problem is an active research area that is problematic even at close range, let alone at a far distance for wind farm occlusion.

Instead, this bachelor thesis shall serve as a foundation for future work by exploring the occlusion problem and giving a good introduction and overview of different depth estimation techniques that can be utilized to solve it. The thesis provides an overview of state-of-the-art depth estimation techniques, elaborating on Computer Stereo Vision, Depth/Structure from Motion, and Monocular Depth Estimation. Furthermore, it introduces a possible new depth estimation technique, DEMs for Depth Estimation, and elaborates how Sky-Segmentation-Algorithms can improve the aforementioned depth estimation techniques.

The thesis defines, disassembles, and analyzes the problem of wind farm occlusion and develops a solution by combining the aforementioned depth estimation techniques as well as developing a new Sky-Segmentation-Algorithm. The ensemble is then implemented in a prototype as a proof of concept.

Using the implemented prototype, a dataset is created with which each depth estimation technique is analyzed and evaluated on its own, as well as in combination with the others according to their applicability and occlusion results for augmented reality on smartphones and wind farm visualization. For this, metrics are introduced, which can be used to analyze the occlusion results numerically.

At last, directions are given for future work.

2 Idea Elaboration

This section lays the foundation for the thesis. Determining which parts of a virtual object should be occluded by the real-world scene requires knowledge about the depth of the real-world objects. Therefore, Section 2.1 elaborates on the most common depth estimation techniques and their advantages and disadvantages, especially in the context of wind farm visualization. This elaboration will mainly focus on computer vision approaches using monocular camera setups since the broad majority of smartphones lack stereo cameras and depth sensors.

Section 2.2 defines and analyzes the occlusion problem for wind farm visualization, splitting the depth estimation into Close- and Long-Distance-Depth-Estimation. At last, Section 2.3 proposes a possible solution, which is later implemented in Section 3, combining different depth estimation algorithms.

2.1 Depth Estimation

In order to visually perceive the three-dimensional space, humans interpret a combination of monocular and binocular depth cues that are present in two-dimensional images and projected on the retinas of the eyes. Each depth cue presents an indication of distance, contributing towards visual depth perception [5]. While binocular cues depend on integrating information from both eyes, monocular cues can also be perceived with only one eye [5].

Correspondingly two research areas have emerged in Computer Vision: Computer Stereo Vision, focusing on binocular cues, and Monocular Depth Estimation, focusing on monocular cues. This division should be regarded carefully since Monocular Depth Estimation only focuses on monocular cues that do not require motion. Here, the depth is inferred only through a single image and not a sequence of images. On the other hand, Computer Stereo Vision yielded the Structure from Motion approach, built upon the same basic principle and math as Computer Stereo Vision but uses a monocular cue requiring motion.

The following chapter introduces the principles of Computer Stereo Vision, transitioning into Structure from Motion, focusing on the two depth cues of binocular parallax and motion/monocular parallax. The subsequent chapter deals with Monocular Depth Estimation, focusing on monocular depth cues and the acquisition of depth estimation from a single image.

2.1.1 Depth/Structure from Motion (SfM) using SLAM

Computer Stereo Vision aims to reproduce the binocular parallax, which refers to “the disparity between two retinal images of a three-dimensional object or scene arising from the slightly different vantage points of the two eyes [...] functioning as a binocular depth cue” [5]. In other words, Computer Stereo Vision compares two digital images of the same scene from different camera angles and computes the disparity, which refers to the difference in coordinates of similar features within two images. In order to acquire the depth of a point inside a 3D scene, epipolar geometry is used to relate one image plane to the other and triangulate the 3D point using the two corresponding 2D points inside the image planes. The absolute distance can be extracted if the camera positions and intrinsic parameters are known or can be derived.

A more advanced technique is the Structure from Motion (SfM) approach, for which a monocular camera setup is sufficient. It builds upon the previously described prin-

ciple but uses the motion/monocular parallax, being a monocular depth cue, which refers to the apparent large displacement of nearby objects and smaller displacement of distant objects triggered by the movement of the observer [5]. So, instead of comparing the images of two different cameras, two frames from different timestamps of the same device are compared. To be successful, the camera is required to move, and the motion must be tracked accurately. If this is given, the depth can be computed the same way as for stereo vision [39]. However, moving objects, as well as inaccurate motion estimates, decrease the accuracy of the depth estimations.

SLAM (Simultaneous Localization And Mapping) [8] is a popular approach to increase motion estimation accuracy. As the name suggests, it tracks the agent while also updating a map for the unknown environment around the agent. SLAM was introduced by the robotics community, and early versions of SLAM algorithms focused mainly on range-sensing techniques, like ultrasound and laser range finders, but also stereo matching [39]. Newer developments also introduced monocular cameras using purely visual feature tracking [39], making it applicable to the majority of smartphones using monocular camera setups. However, in addition to the camera, modern smartphones are equipped with inertial measurement units (IMUs), measuring the smartphone’s acceleration, angular velocity, and orientation [28]. Visual-inertial odometry (VIO) algorithms make use of those IMUs by combining visual tracking with high-frequency IMU measurements to remove the low-frequency drift emerging from visual tracking [39]. Due to the wide use of IMUs in smartphones, those VIO algorithms have been included in recent SLAM developments. Those VIO-enhanced SLAM systems are the foundation on which mobile augmented reality frameworks like ARCore and ARKit are built because they solve the localization problem online, making it the method of choice for real-time augmented reality [39]. Additionally, SLAM allows a wide variety of sensing devices to work together, making it possible to include as many sensing devices in the depth calculations as the smartphone provides [39]. For example, Google states that their ARCore library takes advantage of all available hardware, like ToF cameras, to get the best depth estimation possible [18].

Concluding, VIO-enhanced SLAM Systems would be well suited to be implemented in the wind farm visualization application since it is computationally efficient, and there are already good publicly available frameworks for Android and iOS, namely ARCore and ARKit, which are supported by the majority of newer smartphones. Those frameworks only require a single camera and IMUs for reasonable depth estimations, but additional hardware can easily be added to increase the estimations’ accuracy [3, 18].

However, there are also some drawbacks. Since SfM matches visual features between images to compute the depth, textureless regions or distant regions, which are not accurately captured by the used resolution, cannot be matched between frames. Therefore, SfM cannot compute the depth for those regions. Furthermore, SfM calculates the depth for each pixel separately. Therefore, SfM on its own is not capable of generating a dense, edge-preserving depth map. However, this is needed for immer-

sive augmented reality applications. In 2018, Valentin et al. [44] made a significant breakthrough in cooperation with Google by proposing an implementation that was capable of providing dense and edge-preserving depth maps at low computation on mobile phones by feeding a sparse depth map into a bilateral solver. In 2020 Google extended its ARCore framework by adding a Depth API that was able to provide dense depth maps for occlusion effects [17].

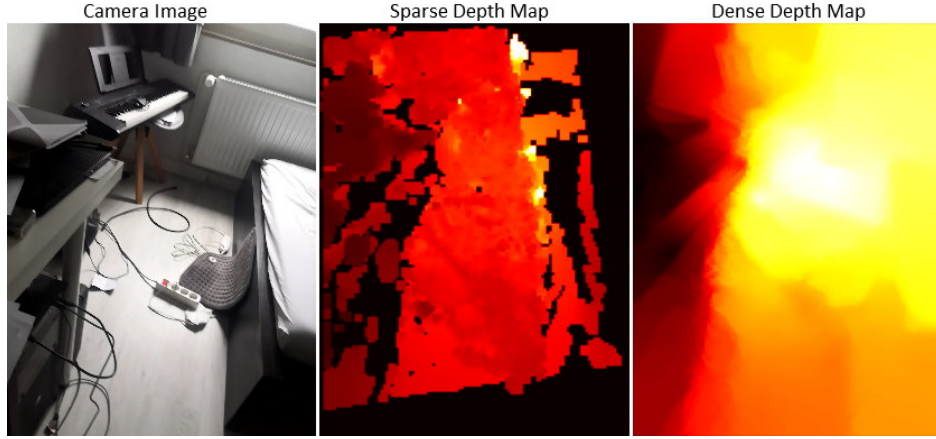


Figure 1: Black regions in sparse depth map represent pixels without depth. Images yielded when using ARCore.

During the process of dense depth map creation, the depth of empty regions inside the sparse depth map is inferred from neighboring pixel depth. Because of that, the depth of regions in which SfM could not estimate any depth is highly dependent on the depth of surrounding pixels. During the evaluation in Section 4.3, it became clear that this behavior is problematic, especially for wind farm visualization. Also, the depth of moving objects cannot be estimated correctly since the movement interferes with the parallax effect.

Furthermore, although ARCore and ARKit support most newer smartphones, some older ones are not supported, decreasing user-friendliness. Additionally, the effective range for online depth estimation is limited due to the lack of computational power. While the maximum detectable distance for ARKit could not be found, ARCore provides a depth estimation of 65.535m, but with significantly decreasing confidence after 8m, according to Google [18]. This is not even close to the depth sensing range required for wind farm occlusion.

2.1.2 Monocular Depth Estimation (MDE)

Monocular Depth Estimation (MDE) aims to infer dense depth maps of a scene from a single image alone. Therefore, monocular depth cues are used, excluding the motion/-monocular parallax, since it requires a sequence of images. To understand MDE and its limitations, it is advisable to look at the used monocular depth cues [5], which are listed below and visualized in Figure 2.

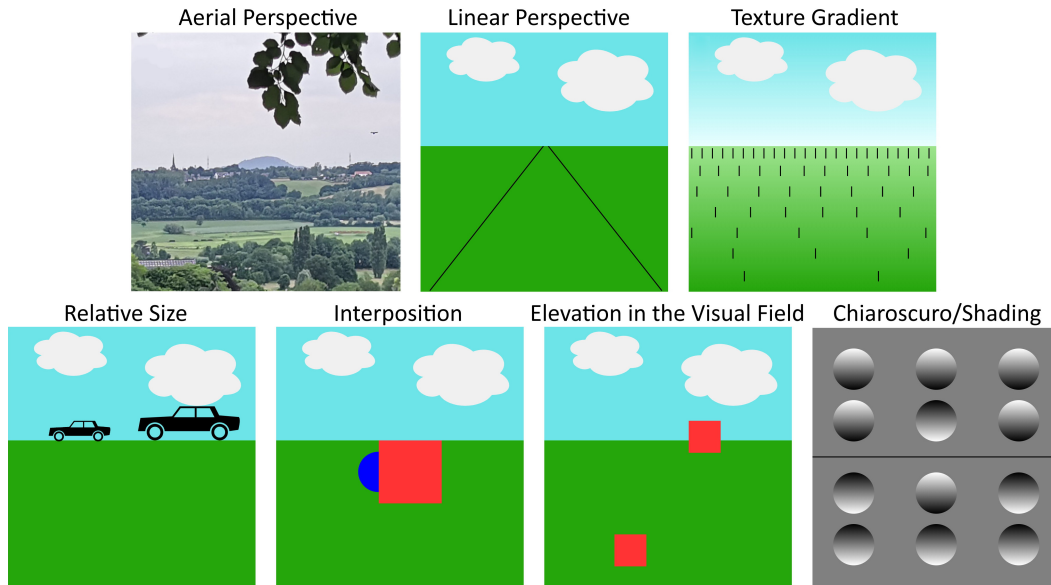


Figure 2: Chiaroscuro/Shading - the bottom image is an inverted copy of the top image (adapted from [5]).

Aerial Perspective: Through the scattering of light by dust and vapor particles in the atmosphere, objects in a great distance have a hazy outline and surface detail, coupled with decreased color intensity or a bluish haze. It serves as a monocular cue and is closely related to linear perspective and texture gradient

Linear Perspective: Describing that an object of fixed size appears to get smaller in the distance such that parallel lines appear to converge towards a vanishing point at the horizon.

Texture Gradient: Change in appearance of the grain or microstructure of a surface. Abrupt changes suggest a contour and gradual change, like objects appearing denser in the distance, being a form of perspective.

Relative Size: The angular sizes of two objects with familiar sizes compared to each other. The greater the distance, the smaller the angular size of an object.

Interposition: Objects partly occluding other objects appear closer. Requires knowledge of the Continuity property: Objects continue to exist behind other objects.

Elevation in the Visual Field: The horizon is typically higher than the foreground, hence the relative height of objects relative to the rest of the visual field serves as a monocular cue.

Chiaroscuro/Shading: The contrast of light and shadow serves as a monocular cue. The human visual system interprets the cue typically under the assumption that the light source is above.

Visual Accommodation: Different distances require the human eye's muscles either to contract to focus on close objects or to relax to focus on distant objects, adjusting the curvature of the crystalline lens and serving as a monocular cue.

All those cues are more challenging to process than simply computing the disparity

between two images. While matching two frames requires simple feature matching and epipolar geometry, MDE requires an understanding of the scene, the objects inside of it, and perspective rules. Relative Size, for example, requires recognizing two objects and attributing them to their known size such that their angular sizes can be compared. While early attempts yielded mixed results trying to solve the problem of MDE by utilizing classical approaches such as working with vanishing points, the direction of straight lines, and image segmentation, in recent years, the research in MDE has been entirely dominated by deep-learning approaches outperforming handcrafted features [14].

However, MDE remains an ill-posed problem [2, 14] and seriously suffers from a lack of scene coverage, scale ambiguities, and translucent or reflective materials [2]. In the case at hand, scale ambiguity is one of the biggest issues. With the only input being the RGB image, the scale becomes arbitrary, making it extremely hard for a deep neural network to estimate absolute depth values, see Figure 3.

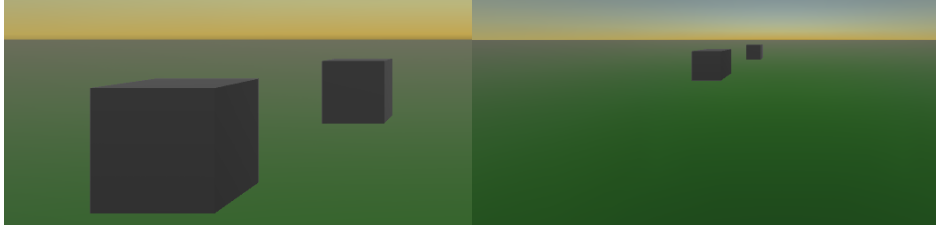


Figure 3: Scale is arbitrary. Both images show the same scene; only the FOV of the camera has changed, demonstrating that the absolute depth cannot be estimated using solely RGB images.

Instead, deep neural networks are usually trained to estimate the relative distance of the objects inside the scene from each other [9]. To put it in other words, the neural networks are trained only to be able to tell which objects are behind others but cannot output the distance of an object toward the camera. This is a big issue since the wind turbine shall be occluded by comparing its absolute distance to the camera with a computed depth map deciding which pixels should occlude the wind turbine. Yet, comparing the absolute distance of the wind farm with the estimated relative values from MDE is not possible. This problem of relative values is one reason why it is said that the depth is not really inferred but rather hallucinated by the network [39].

An additional problem of using deep neural networks is that they are computationally expensive, making them unsuitable for estimating (or hallucinating) the depth for all 30 frames each second, especially when deployed on a smartphone. Hence, this approach is problematic because of the high computational costs and the problem of relative depth values. However, MDE neural networks can provide edge-preserving dense depth maps without the need for visual feature matching but solely from a single RGB image, capturing object boundaries more faithfully than SfM.

Several publicly available datasets provide training and test data for neural network training for MDE. The NYU Depth v2 [37] dataset consists of around 120k images and

corresponding depth maps of indoor scenes having an upper bound of 10 meters [2]. The KITTI [16] dataset consists of stereo images and corresponding 3D laser scans of outdoor scenes, which were captured using equipment mounted on a moving vehicle while driving through urban areas. The provided depth maps have an upper bound of 80 meters and a very low density with lots of missing data [2]. Other datasets are the ScanNet [6] dataset consisting of various indoor scenes, and the Make3D [34, 35] dataset consisting of urban and natural areas using laser scanning.

2.1.3 Digital Elevation Model (DEM) for Depth Estimation

The Digital Elevation Models (DEM) for Depth Estimation approach is not based on computer vision since it does not use visual features. Instead, it uses the GPS location and the estimated view direction to place the user inside a 3D representation of their real-world surroundings. Assuming that there exists such a 3D representation that is precise enough and under the assumption that the location and view direction can be estimated precisely, the depth can be efficiently extracted at runtime from the Z(depth)-buffer since the depth is needed anyway during rendering to decide which virtual objects should be occluded by others. However, the accuracy of the extracted depth values depends on the precision of the GPS localization, the view direction estimation, and the 3D representation.

GPS and View Direction: Once initialized correctly, the position and view can be precisely adjusted according to the device’s movement using the IMUs and visual feature tracking. Furthermore, it is not expected from the user to move long distances, so a large drift is not expected. Therefore, theoretically, the GPS and compass are only required during the initialization.

According to Google, GPS can localize the horizontal position within a 5-10m radius (the altitude is even worse) and the view direction within 30-45 degrees [36], which is supported by [23, 45]. To improve this and make global localization ready for augmented reality applications, Google added a Visual Positioning System (VPS) in their ARCore Geospatial API [19]. This VPS uses the current camera image and matches it with the Street View data Google has acquired in the last decade. Out of this match, the location and orientation are calculated much more accurately “than what was previously possible with GPS alone” [19]. The extent of the improvement, however, is not disclosed. Although this imprecision is a critical factor, this bachelor thesis does not focus on this overall visualization problem of augmented reality applications since the focus of this thesis lies on depth estimation. Furthermore, the farther the virtual objects are away, the less influence the inaccuracies have.

3D Representation: In order to extract the depth information according to the user’s position and a 3D representation of the user’s surroundings, this 3D model must be created beforehand. For small regions, models can be created by hand, but for large-scale applications, like visualization of wind farms, where the representation needs to contain large areas of a few kilometers, manual modeling becomes infeasible.

Therefore, preexisting models are required, which should be selected according to their availability and accuracy. For the purpose of this thesis, the focus will lie mainly on Digital Elevation Models (DEMs) and how they can be enhanced.

In this thesis, the term Digital Elevation Model (DEM) will be used to describe the union of Digital Terrain Models (DTMs) and Digital Surface Models (DSMs), which are 3D computer graphics representations of elevation data. While DSMs describe the height of the surface, including objects like buildings and trees, DTMs only describe the terrain beneath all those objects (bare-earth elevation). This specification is important since there is no universal usage of the term DEM. While DTMs may not be suitable for flat regions like Aachen, they may be good representations for mountainous regions. On the other hand, DSMs may be outdated since objects on the terrain typically change faster than the terrain itself.

Overview of available DEMs: Probably, the most popular near-global DEM is provided by the US government. It was derived from radar data obtained during the Shuttle Radar Topography Mission (SRTM) [12] in February 2000. The data is available at a resolution of 1 arc-second or about 30 meters. The vertical accuracy is not given. Since the radar cannot pass through buildings or sometimes even trees, the SRTM DEM is a DSM. Due to its near-global scale and free availability, it has been widely used.

Another global DEM is the WorldDEM [33], which provides a resolution of 0.4 arc-seconds or 12 meters with a relative vertical precision of less than 4 meters (WorldDEM Neo provides a resolution of 5 meters). It was obtained through two German radar satellites between 2010 and 2013 and contains both a DSM and a DTM. However, this DEM is not freely available and requires license fees.

For a more extensive overview of global DEMs, see Figure 1 in the paper of Hawker et al. from 2018 [21].

Fortunately, DEMs at a global scale are not required for this thesis. The European Union provides the free-to-use EU-DEM v1.1 [4] covering complete Europe. It was derived out of the SRTM and ASTER GDEM [1] data, providing a resolution of 25m and vertical accuracy of $(+/-)7m$ root mean squared error.

Narrowing the region of interest down to Germany, even better resolutions become available. The Federal Agency for Cartography and Geodesy of Germany provides free DTMs with a resolution of 200m (DGM200) and 1000m (DGM1000) and a fee requiring DTM with a resolution of 5m (DGM5)¹, all with a vertical and horizontal accuracy of $(+/-)0.3-1m$. Additionally, they provide a DSM named DOM1² with a resolution of 1m,

¹<https://mis.bkg.bund.de/trefferanzeige?docuuid=86840e2a-4e98-402d-be12-f418695c7369>

²<https://gdz.bkg.bund.de/index.php/default/digitale-geodaten/digitale-gelandemodelle/digitales-oberfaechenmodell-dom1.html>

horizontal accuracy of $(+/-)0.5-1m$ and vertical accuracy of $(+/-)0.1-1m$. All covering complete Germany. Unfortunately, the DOM1 can only be accessed by federal agencies.

While DEMs are available for entire Germany with extremely high resolutions and accuracies, those are usually kept under lock and key or are expensive. Fortunately, the “open data policy” has gained popularity in Europe. While neighboring countries like England and Slovakia have already made their data publicly available, some federal states in Germany have started to go the same way. NRW, being a pioneer in this area, provides different topographic datasets under the “Data license Germany - Zero - Version 2.0”³, which permits any use without restrictions or conditions. One of those datasets being the “3D-Messdaten”⁴ dataset describing the terrain and surface of NRW as 3D point clouds. This LiDAR data is obtained through airborne laser-scanning (ALS) and provides a horizontal accuracy of $(+/-)30cm$ and a vertical accuracy of $(+/-)15cm$ (twice the standard deviation). According to the description, it is also possible to generate DSMs as well as DTMs out of this dataset.

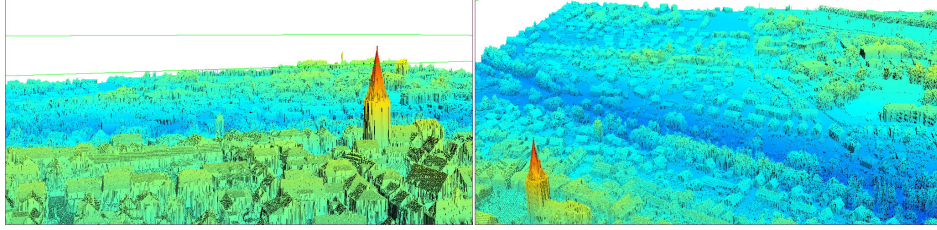


Figure 4: “3D Messdaten”/LiDAR example region when reading the data with a tool named “just a little LAZ and LAZ viewer” with “triangulate” option activated.

Additionally, the state of NRW provides access to the “3D-Gebäudemodelle”⁵ data, which is constructed out of the “3D Messdaten” the “ALKIS” and “ATKIS” datasets completely modeling NRW and its buildings either without roofs (LoD1) or with automatically generated roofs (LoD2). Roofs of more complex buildings were modeled differently. The vertical accuracy of the LoD2 version amounts to $(+/-)1m$, and the horizontal accuracy depends on the “3D Messdaten”, “ALKIS”, and “ATKIS” data. All of this is also distributed under the “Data license Germany - Zero - Version 2.0”.

While every federal state in Germany maintains its own corresponding “3D Messdaten” dataset and “3D-Gebäudemodelle” data, only a few of them follow the model of NRW. Some states make their data publicly available, others provide access on request, sometimes a handling cost must be paid, and some states sell their data for high costs. Figure 5 gives a brief overview of the different handling among the federal states.

In conclusion, this depth estimation approach is highly dependent on the precision of the localized position and the estimated view direction, as well as the precision of

³<https://www.govdata.de/dl-de/zero-2-0>

⁴https://www.bezreg-koeln.nrw.de/brk_internet/geobasis/hoehenmodelle/3d-messdaten/

⁵https://www.bezreg-koeln.nrw.de/brk_internet/geobasis/3d_gebaeudemodelle/

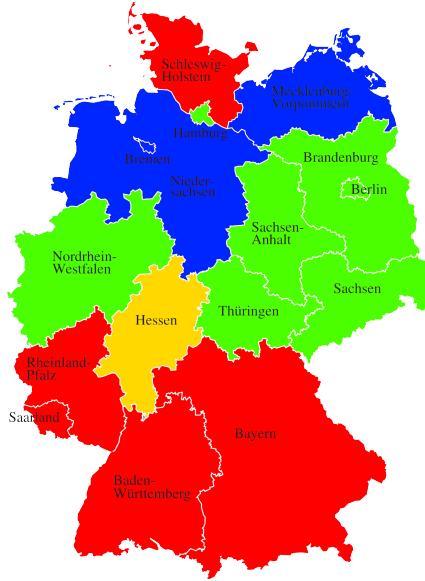


Figure 5: Overview of data availability for each of the federal states of Germany⁶. Green: State provides open access to “3D Gebäudemodelle” and “3D Messdaten” datasets, or data can be requested free of charge. Yellow: “3D Gebäudemodelle” dataset is open access, but the “3D Messdaten” dataset needs to be requested and requires a small fee as compensation for expenses. Blue: “3D Gebäudemodelle” and “3D Messdaten” datasets can be requested, but no information about the costs is given. Red: “3D Gebäudemodelle” and “3D Messdaten” datasets must be bought.

the 3D representation. Therefore, the approach lacks precision but may be a viable approach for great distances. Additionally, after a 3D representation has been created, it is fast to compute since the GPS and view direction need to be evaluated nevertheless, and the depth can be acquired during the rendering process. Furthermore, the only distance/range restriction is the size of the 3D representation.

2.1.4 Sky Segmentation

This approach uses semantic image segmentation to detect sky pixels and set their corresponding depth to be greater than that of the wind farm. This assures that no sky pixels occlude the virtual objects. Although this approach may neither be suitable for close-distance occlusion nor indoor occlusion, it is certainly a valid option for long-distance occlusion if the virtual object is expected to rise up to the sky, like a wind turbine.

At first glance, Sky Segmentation seems very simple. However, due to a large variety of lighting and weather conditions, the problem turns out to be rather complicated [29]. Some problematic scenarios are blue sky partially occluded by white or grey clouds, sunrise, and sunset. A recent approach was published by Nice et al. in 2020 [29].

⁶Picture taken and modified from <https://pixabay.com/de/vectors/karte-deutschland-bundeslaender-606538>

They used a convolutional neural network in order to choose one of three segmentation techniques and its parameters tailored to the current image. The three techniques are mean-shift segmentation [15], K-Means clustering [13, 26], and Sobel filters/operators [24]. In the end, they reached an accuracy of 88% on the Skyfinder [27] dataset performing better than other published techniques [29]. While mean-shift segmentation is (relatively) computationally expensive, K-means clustering and Sobel filters can be computed very fast and could even be run in real-time on a smartphone with 30fps. However, the hardest part of the task is accomplished by the neural network, which adapts the approach to different lighting and weather conditions in the images.

In conclusion, while the underlying techniques are very efficient in detecting pixels that should not occlude the virtual object, their performance is situational and rely on the chosen hyperparameters, which need to be adapted to different lighting and weather conditions by a complex neural network [29]. Sky Segmentation is a promising approach to partially solve occlusion in real-time but must be complemented with another approach to solve occlusion for pixels not belonging to the sky.

2.2 Problem Statement | The Occlusion Problem

Having introduced several methods that can be used to solve occlusion, this section discusses which of those methods might be suitable for the problem at hand of occluding wind farms at a far distance. For simplicity, from now on and for the rest of the thesis, the number of virtual wind turbines is reduced to one, which will be referenced as the Scene-Object. When solving the occlusion problem for wind turbines, there are three

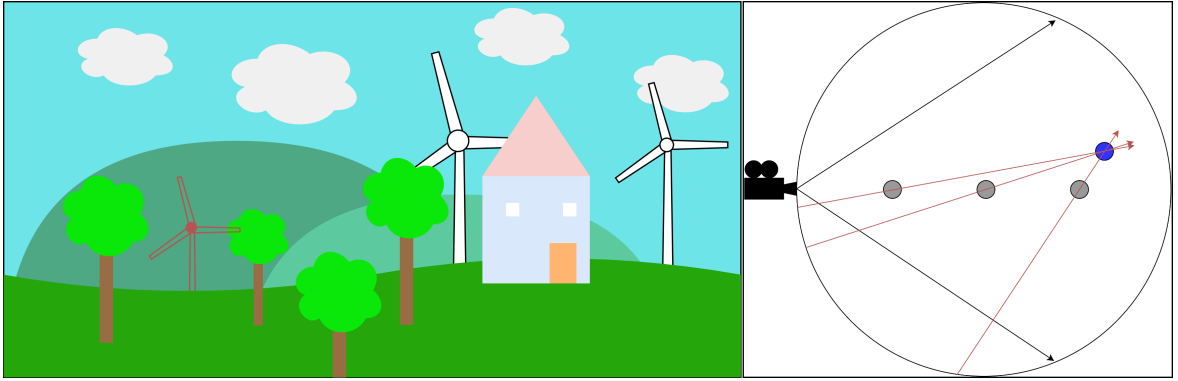


Figure 6: Left image: Occlusion. Left wind turbine fully occluded. Right wind turbine only occludes sky pixels. Middle wind turbine occludes sky pixels and hill pixels. Right image: Demonstrating the parallax effect: the farther an object (gray) is away from the camera, the more the camera must move in order to bring the object in front of the Scene-Object (blue).

cases to be considered, as visualized in Figure 6:

1. the Scene-Object is fully occluded,
2. the Scene-Object is partly or fully visible but does only occlude sky pixels,

3. the Scene-Object is partly or fully visible and occludes non-sky pixels.

While the first two scenarios require no depth estimation techniques and a less complex Sky-Segmentation-Algorithm suffices, the third requires depth estimation since it is not trivial to differentiate between non-sky fore- and background pixels. Therefore, solving the third scenario is a complex problem.

While the distance of the Scene-Object to the user’s current location is given, it is sufficient to know whether or not the depth of a pixel exceeds the distance of the Scene-Object, resulting in a Boolean array: The occlusion map. To derive and complement the occlusion map, two possible approaches are as follows:

1. Forward-Mode: Set all pixels as default to being occluded by the Scene-Object and apply methods to find pixels that should occlude the Scene-Object (detect foreground pixels).
2. Backward-Mode: Set all pixels as default to occlude the Scene-Object and apply methods to find pixels that should not occlude the Scene-Object (detect background pixels).

In this work, the Forward-Mode will be explored since most depth-inferring techniques struggle with growing distance, making it easier to infer the depth of pixels in front of the Scene-Object. Additionally, it is easier for the user to interpret a scene with the Scene-Object falsely occluding non-virtual objects than the other way around. Combining fore- and background detection leads to some complications since then a pixel could be classified as foreground and background at the same time.

Hence, the used occlusion map is defined as follows, with False being the initialization value, indicating that the pixel is not in the foreground:

$$Occlusion_Map(i, j) = \begin{cases} True, & \text{if pixel}(i, j) \text{ is in front of Scene-Object} \\ False, & \text{else (background/unknown pixel)} \end{cases} \quad (1)$$

To solve the occlusion problem, the Occlusion-Map has to be correctly inferred for each frame, as the relative depth can change dynamically if, e.g., moving objects might move in front of the camera. On the other hand, the application needs to solve the occlusion problem for large distances requiring a lot of computational power since inferring depth at long distances requires greater resolutions to stay accurate, increasing computational complexity. None of the previously elaborated methods satisfies all requirements for wind farm occlusion. Although SfM is computationally efficient, modern libraries like ARCore only provide estimations up to 65.535m with significantly decreased precision after 8m [18]. The monocular depth estimations, on the other hand, are highly computationally expensive [44], making an analysis of every frame impossible. DEM for Depth Estimation relies entirely on the accuracy of the used DEM (which will never be a perfect model of the real world) and the GPS-based positioning of the user, both lacking precision in order to be able to provide accurate occlusion results. The Sky-Segmentation-Algorithm is the only method that can

be applied efficiently for every frame and every distance, but it is unable to distinguish non-sky fore- and background pixels. Hence, the problem cannot be solved by one of the described methods alone. Because of this, this thesis proposes to split up the Depth-Infering-Problem into Close-Distance-Depth-Estimation (Close-DDE) and Long-Distance-Depth-Estimation (Long-DDE).

Close-Distance-Depth-Estimation (Close-DDE): At close distances, a slight camera movement, as well as moving objects, have a substantial influence on the scene because of the parallax effect. Camera movements may cause a change in perspective or could bring static objects between the camera and the Scene-Object. Moving objects, like people or cars, may move in front of the camera, also occluding the Scene-Object. Hence, the depth needs to be calculated for every frame since the scene can change significantly between frames. On the other hand, the application has reduced computational complexity since the depth must only be inferred at a close distance requiring a smaller resolution, making the computation for every frame feasible.

Long-Distance-Depth-Estimation (Long-DDE): At long distances, the camera movement, as well as moving objects, have a minor influence. At long distances moving objects are scaled down to a negligible proportion of the screen since they are typically not extremely big. Additionally, larger camera movements are necessary to bring far away static objects between the camera and the Scene-Object, see Figure 6. Since only small movements are expected from the user, the distant three-dimensional world can be considered as a two-dimensional image plane, omitting any change in perspective. Therefore, after computing the depth around the Scene-Object once, it can be reused and adjusted to the camera movement in subsequent frames, reducing the computational cost of recomputing the depth for every frame. Occasionally, the depth map can be recomputed when needed.

2.3 Solution

Because of the above-described differences between Close- and Long-DDE, it is advantageous to solve them separately and combine the resulting close and long occlusion maps into a combined occlusion map afterward. Pursuing this concept, a computationally inexpensive algorithm is needed for Close-DDE since the depth must be estimated for each frame. Since it is possible to reuse the Long-DDE depth in subsequent frames, reducing the computational costs, the solution for Long-DDE can be chosen to be more computationally expensive. However, it remains worth considering outsourcing the Long-DDE to a server. In the end, the two separately derived occlusion maps, the close- and long-distance occlusion map, are combined, complementing each other with regard to their foreground classification, while background/unknown pixels are overwritten if one of the approaches has classified the pixel as foreground. Therefore, each approach can only add new foreground pixels and is not able to reverse a foreground pixel classification in this approach.

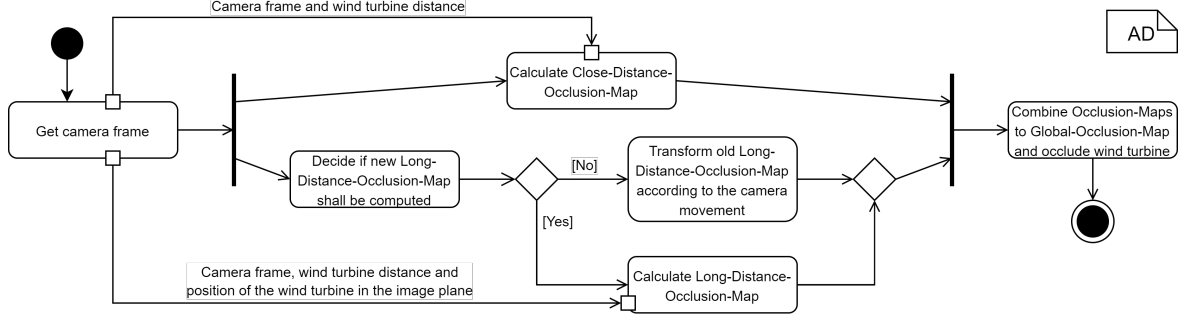


Figure 7: Activity diagram of how the combined conclusion map is derived for every frame.

Close-DDE: Structure from Motion (SfM) using VIO-enhanced SLAM

To solve the Close-DDE, this thesis proposes to use the SfM approach using a VIO-enhanced SLAM system since it is the most computationally inexpensive depth estimation approach for close distance, and there are popular libraries available like ARCore (Android) and ARKit (iOS) providing such Depth APIs ensuring that this approach works for both platforms. While ARKit has not been explored in this thesis, ARCore uses a 120×160 resolution input image and available depth sensors to efficiently compute a dense depth map containing depth values from 0-65.535m fluently for 30FPS. The SfM occlusion map can be derived by comparing the resulting SfM depth map with the wind turbine distance. However, SfM algorithms cannot accurately estimate the depth of moving objects since they depend on the motion/monocular parallax.

Long-DDE: DEM and Monocular Depth Estimation

To solve the Long-DDE, this thesis proposes to use a combination of the DEM for Depth Estimation, and Monocular Depth Estimation (MDE) approaches.

While the depth can be very efficiently extracted out of 3D representations, available models and the GPS positioning are too imprecise, making this approach unsuitable for Close-DDE. Even in Long-DDE, those inaccuracies may have strong negative effects on the resulting combined occlusion map. Instead of using DEM for Depth Estimation directly for the occlusion map, it is used to help solve the relative values problem of MDE. Despite its inaccuracies, the DEM depth data is a good indicator for regions of foreground pixels. Those regions can be mapped onto the MDE depth map indicating which relative values of the MDE depth map are likely to describe pixels in front of the wind turbine. Out of those relative values, a threshold can be computed, which can then be used to classify each relative MDE depth value into foreground or background/unknown pixel, bypassing the limitation of having no absolute but relative depth values.

Another limitation of the MDE approach is the resolution that can be fed into the neural network. State-of-the-art depth estimation neural networks are usually computationally expensive [2, 32]. Hence the resolution of the input image is usually very low. Because of that, the camera image must be downsampled to fit the expected resolu-

tion if the depth shall be estimated for the entire image, losing features in the distance and limiting the range of the depth estimation. However, estimating the depth of the entire camera image is not necessary. It is sufficient to estimate the depth around the wind turbine. Therefore, it is proposed to locate the wind turbine in the image plane and cut out a region around it, which is then fed into the neural network. The size of the region can be adjusted according to the distance of the wind turbine, more or less like a digital zoom. This region will be referenced in the following as the Cutout region. Additionally, it is beneficial to rotate the Cutout region according to the wind turbine such that the horizon is horizontal and the wind turbine is facing upward. This way, the neural network is less prone to wrong depth estimations evoked by orientation issues.

As described above, the estimated MDE depth map from the Cutout region can then be classified into foreground and background/unknown pixels using the threshold computed with the help of the DEM depth map, as described above. Figure 8 visualizes the process.

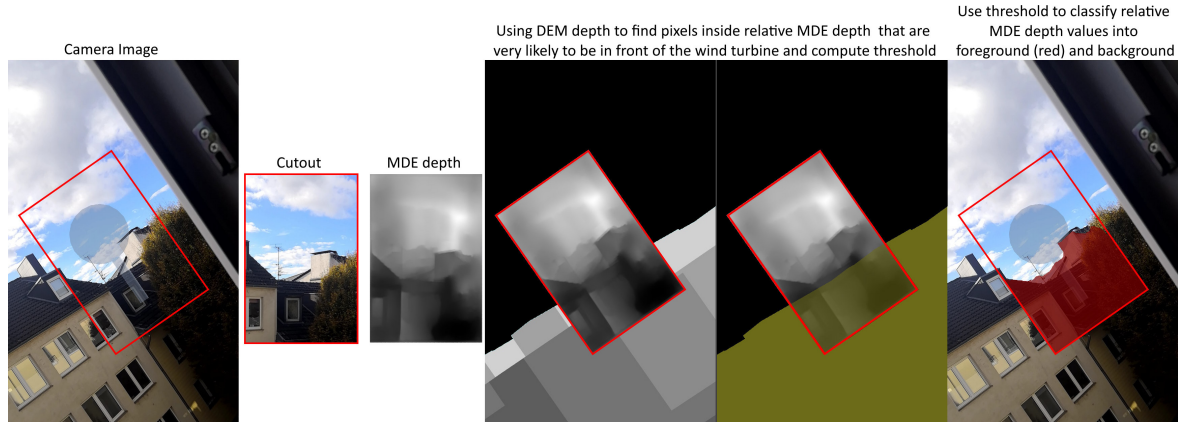


Figure 8: Visualization of the process which derives the long-distance occlusion map. 1. Region around the wind turbine inside the camera image is selected. 2. Region is extracted and rotated according to the wind turbine and fed into the neural network. 3. The MDE depth map is placed back into the image plane. 4. The DEM depth map is used to select a foreground region of relative depth values inside the MDE depth map. 5. Out of those values, a threshold is computed, which is used to classify each pixel according to its corresponding MDE depth value into fore- or background/unknown.

Optional: Sky-Segmentation-Algorithm

Optionally, a Sky-Segmentation-Algorithm can be applied to correct misclassified sky areas. However, an implemented Sky-Segmentation-Algorithm will classify each pixel into either background or foreground/unknown, following the Backward-Mode from Section 2.2. This reversed classification is problematic when combining it with the other occlusion results (see Section 3.2.4) since a pixel can then be classified into foreground and background at the same time, making a decision necessary.

3 Implementation and Insights

This Section describes the implementation of the previously defined solution (see Section 2.3).

Overall, this thesis develops a prototype that encompasses Structure from Motion (SfM) and the inclusion of DEM for Depth Estimation inside of Unity (see Section 3.1), being a cross-platform game engine with which AR content can be easily created since it provides easy access to the features of ARCore and ARKit through the AR Foundation Plug-in [43]. Unity was previously identified as a suitable framework for the project [30], as it allows cross-platform development for Android and iOS [43]. Furthermore, the thesis explores the additional application of neural networks for Monocular Depth Estimation (MDE) and the application of Sky Segmentation, which are performed outside of Unity using Python (see Section 3.2) for explorative reasons. While the results of this thesis (see Section 4.7) suggest that the use of neural networks for MDE can significantly improve the occlusion, practical questions must be answered, such as whether the neural network is applied on a server or can be efficiently integrated into the app. The implementation and comparison of these practical problems are outside the scope of this thesis.

Unity Environment

The prototype uses the Unity Editor v2021.3.6f1 with Android NDK, Android SKD Build Tools, and Android SDK Platforms installed. Furthermore, the AR Foundation Plug-in v.5.0.2, which is the newest stable version, was used. It was crucial to use AR Foundation v.5 since this is the first version supporting ARCore v.1.31, which is again the first version that uses all 16 bits of the estimated depth providing 0-65.535m depth estimation instead of 0-8.191m [18]. This ARCore version was released in May 2022, shortly before this bachelor thesis, and during the bachelor thesis, the stable version of AR Foundation v.5 was published. Previous ARCore versions used only 13 of the 16 bits of its estimated depth values, setting unwanted bits to zero, requiring additional computations to provide an 8.191m instead of 65.535m depth estimation [40, 41]. However, the changelogs of AR Foundation state that AR Foundation v.4.1.0-preview.13, released on 2020-11-09, uses all 16 bits of the ARCore depth estimation, although the used ARCore version does not support this⁷. This could not be verified. AR Foundation v.5.0.2 also uses ARKit 6 [41, 42]. However, the ARKit version is irrelevant to this thesis since the prototype is developed only for Android. Additionally, the range of ARKits depth estimation could not be found [3]. Therefore, this prototype has limited expressiveness with regard to the effectiveness of the Structure from Motion algorithm on iOS devices.

⁷<https://docs.unity.cn/Packages/com.unity.xr.arcore@4.1/changelog/CHANGELOG.html>

Python Environment

For this bachelor thesis, a Python environment was created using conda, a cross-platform package and environment manager. The environment uses Python v.3.8.5 with the following packages installed: Jupyter, TensorFlow-gpu, Matplotlib, Pillow, pandas, Keras, scikit-learn, scikit-image, pydot, tqdm, seaborn and OpenCV 2. The environment can be recreated using the following three commands:

```
1 $ conda create -n depthEnv jupyter tensorflow-gpu matplotlib pillow pandas keras  
    scikit-learn scikit-image pydot tqdm seaborn  
2 $ conda activate depthEnv  
3 $ conda install -c conda-forge opencv
```

3.1 Unity | Depth from Motion and DEM for Depth Estimation

The Unity application implements the Depth from Motion approach, solving the Close-DDE, and the first part of the Long-DDE using the DEM for Depth Estimation approach. Apart from that, the application also locates the wind turbine in the image plane, saves all required data on the device, and provides utility functionalities making it possible to record different scenes from different locations in Aachen, which will be used to evaluate the utilized algorithms.

Unity Project Setup

As described above, this part of the prototype is developed using Unity. Therefore, a new Unity project is created using the “AR Core” template provided by Unity. This template sets some necessary settings automatically, saving some work. The new project already has an older version of AR Foundation installed. In order to update AR Foundation, the old version must be removed, and the new version must be installed from the “Unity Registry” using the Package Manager of Unity. The “Add package by name” function allows the installation of packages with a specified version. The AR Foundation package is addressed using the unique key “com.unity.xr.arfoundation” and the desired version, here, “5.0.2”. In addition to AR Foundation, the ARCore XR Plug-in must be installed with the unique key “com.unity.xr.arcore” and version “5.0.2”.

After that, some adjustments must be made to the Player Settings for Android devices under Project Settings. Under * → Other Settings → Rendering, the “Multithreaded Rendering*” must be deactivated since this option can cause serious problems working with ARCore XR Plug-in. Additionally, under * → Other Settings → Script Compilation, the “Allow ‘unsafe’ Code” option must be activated since unsafe code is needed during data extraction from the Unity application to the Python environment. Furthermore, the “Default Orientation*” is set to “Portrait” under * → Resolution and Presentation → Orientation, disabling the automatic rotation of the screen content when the device is rotated. This erases the need to check for device orientation when displaying the GUI elements and rendering the textures. Otherwise, the GUI elements and texture would need to be oriented accordingly. At last, the ARCore XR Plug-in must be activated

under Project Settings →XR Plug-in Management →Android →Plug-in Providers. After deleting all GameObjects from the current Scene, the project is clean and set up.

Basic Structure of AR Foundation Projects

The linchpin of each AR Foundation application is the “AR Session” and “XR Origin” GameObjects, which are only allowed once inside each Scene.

The “AR Session” manages and controls the lifecycle of the AR experience with its attached Components “AR Input Manager (Script)” and “AR Session (Script)” enabling or disabling the augmented reality experience on the target platform [41]. With the attached “AR Session (Script)” Component, the “Tracking Mode” can also be set to either “Position and Rotation” or “Rotation Only”. The latter could increase the accuracy of the depth estimations but would restrict the user’s movement.

The “XR Origin” serves as a transformer converting trackable features, like planar surfaces and feature points, from the AR device session space into their final position, rotation, and scale in the Unity world space [41]. Attached to the “XR Origin” GameObject is the “Camera Offset” GameObject to which the “Main Camera” GameObject is attached, being a Unity Camera and the virtual equivalent of the device camera. This “Main Camera” has three essential Components attached to it. The first two, “AR Camera Manager (Script)” and “AR Camera Background (Script)”, provide the virtual camera with the camera texture. The third Component, “Tracked Pose Driver (Input System)”, passes the device pose information to the “AR Input Manager (Script)” [41].

3.1.1 Structure from Motion (SfM) | Close-DDE

After adjusting the project according to the AR Foundation guidelines, the wind turbine 3D model can be added to the project. The very basic model contains the wind turbine pole being a cylinder with a radius of 10m and height of 160m, the wind turbine rotor being also a cylinder with a radius of 60m, and two empty objects representing the position of the tip of the wind turbine and the position of the bottom of the wind turbine. The rotor was visualized using a cylinder, modeling all possible positions of the blades instead of single blades to replace their rotation animation (see Figure 9).

By default, the far clipping plane of the “Main Camera” is set to 20m. Since the wind turbines will be placed farther away, the clipping plane needs to be adjusted according to the maximum distance at which the user should be able to see the virtual wind farm.

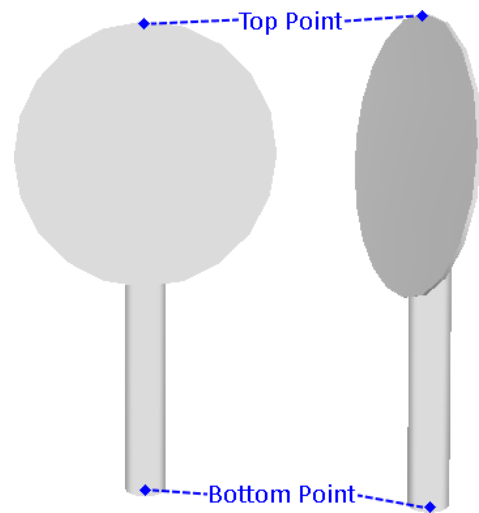


Figure 9: Wind turbine model.

To enable occlusion, the “AR Occlusion Manager (Script)” Component needs to be added to the “Main Camera”. This manager passes the depth values as a texture to the “AR Camera Manager” and “AR Camera Background” Components. The “AR Occlusion Manager” offers various settings, influencing the quality of the resulting depth estimations. In this thesis, the “Environment Depth Mode” was set to “Best” and the “Temporal Smoothing” was activated, which allows the environment depth to be smoothed over time [41]. This makes the depth estimation more robust and seems necessary for good results. The “AR Occlusion Manager” also provides options to activate “Human Segmentation”. However, this feature is not available for ARCore but only for ARKit [41].

Only these few steps are needed to successfully enable the occlusion using AR Foundation. However, the “AR Occlusion Manager” is not intended to enable occlusion at long distances. Because of that, the system assigns each pixel a depth value between 0 and 65.535m, occluding any virtual object farther away completely. Additionally, since the underlying algorithm used by the system is based on the SfM approach and since the system uses a resolution of 120×160 during depth estimation, the system’s confidence for great depth estimations decreases significantly [18], also because of the dense depth map construction (see Section 2.1.1).

Therefore, a threshold must be set that trades off the range of the depth estimation with its estimation confidence. A suitable value for the threshold is derived in Section 4.3.1. In order to apply the threshold to the application, the shader, which handles the occlusion and is provided by the ARCore XR Plug-in, must be modified, and the “AR Camera Background” manager must be adjusted to use the new shader. The provided default shader “ARCoreBackgroundAfterOpaaques.shader” (see Shader 1) is written in OpenGL ES 3.0, a subset of OpenGL specifically for embedded systems (ES) like smartphones. Shaders written in other languages do not work on Android smartphones. The “ARCoreBackgroundAfterOpaaques.shader” contains two passes. The first pass shall discard any fragments (pixel candidates) as early in the rendering process as possible if they are already known to be occluded by the camera pixels. This shall improve the performance of the application since fewer fragments must be tested during the z/depth test. The second pass then draws the camera texture to the screen according to the distance of the virtual object and the estimated pixel depth.

However, the first pass was not functioning correctly, which probably went unnoticed due to the lack of visual difference. The pass used undefined variables and functions, and a 4-dimensional vector provided by OpenGL ES was assigned to a 2-dimensional vector without casting. These errors individually caused the pass to break, resulting in it having no impact during the rendering process and being skipped. The issue and a suggested improvement were reported, and a fix is pending⁸. Once the shader was

⁸<https://github.com/Unity-Technologies/arfoundation-samples/issues/1017>

fixed (see Shader 2), adapting it to only occlude virtual objects when the environment depth is smaller than the threshold was relatively straightforward (see Shader 3).

3.1.2 DEM for Depth Estimation | Long-DDE/Region Selection

Although there are very precise DEMs modeling all of NRW using LiDAR and “ALKIS” data (Section 2.1.3), they require a lot of preprocessing before they can be imported into Unity. Since creating Aachen as a precise 3D model is not the focus of this thesis, the SRTM DEM has been used instead. This shall serve as a proof of concept that DEMs are able to improve the depth estimation by helping to solve the relative depth problem of MDE neural networks (see Section 2.3).

The advantage of using SRTM lies in the availability of software supporting the creation of 3D models using SRTM data. Here, Blender⁹, a free and open-source 3D computer graphics software, was used in combination with the freely available Blender GIS Plug-in¹⁰. Using both, creating a 3D model of Aachen using the DSM SRTM requires only a few steps. Additionally, the Blender GIS Plug-in provides an option to automatically model all buildings of any selected region using the building outlines and the number of floors, all provided by OpenTopography. The height of each building is estimated using the number of floors and a default floor height value, which can be set by hand. The steps of this process are specified below:

First, the Blender file must be created by selecting File → New → General inside Blender. After that, the camera and light must be deleted to not interfere with the camera inside the Unity project. After that, the Blender GIS Plug-in is used to import the Basemap (see Figure 10): GIS → Web geodata → Basemap → OK. This opens up a world map in which an address can be searched using the shortcut G, and a region can be selected by pressing the key E. The Basemap is imported into the blender file showing the selected region’s satellite image.

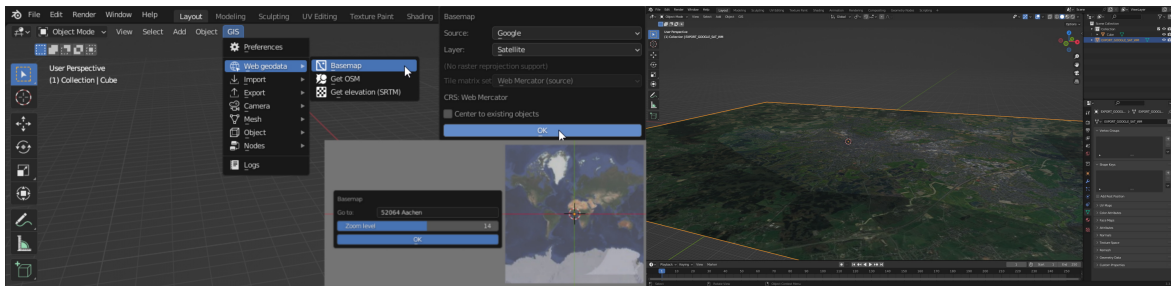


Figure 10: Getting the Basemap using Blender GIS Plug-in.

Then, the SRTM elevation data with a resolution of 30m×30m can be mapped onto the Basemap (see Figure 11) by selecting the “EXPORT_GOOGLE_SAT_WM” object and using the Blender GIS Plug-in again: GIS→Web geodata →Get elevation (SRTM)

⁹<https://www.blender.org/>

¹⁰<https://github.com/domlysz/BlenderGIS>

selecting OpenTopography as Server and entering an API Key, which can be requested from OpenTopography free of charge.

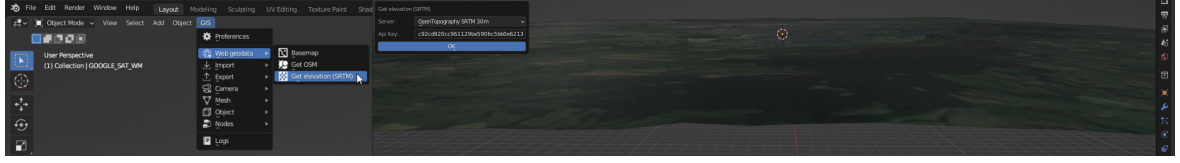


Figure 11: Mapping SRTM elevation data (30m×30m resolution) onto the Basemap using Blender GIS Plug-in.

At last, the building data can be imported (see Figure 12) by selecting the “EXPORT_GOOGLE_SAT_WM” object, after which the Blender GIS Plug-in can be used again, GIS → Web geodata → Get OSM, opening up another window. There the tab “Ways” must be selected, and “Elevation from object” must be activated and set to the “EXPORT_GOOGLE_SAT_WM” object. “Building extrusion” must also be activated. Below, the “Level height” can be set, which is used to calculate the height of each building. This operation can take a few minutes, depending on the size of the selected region.

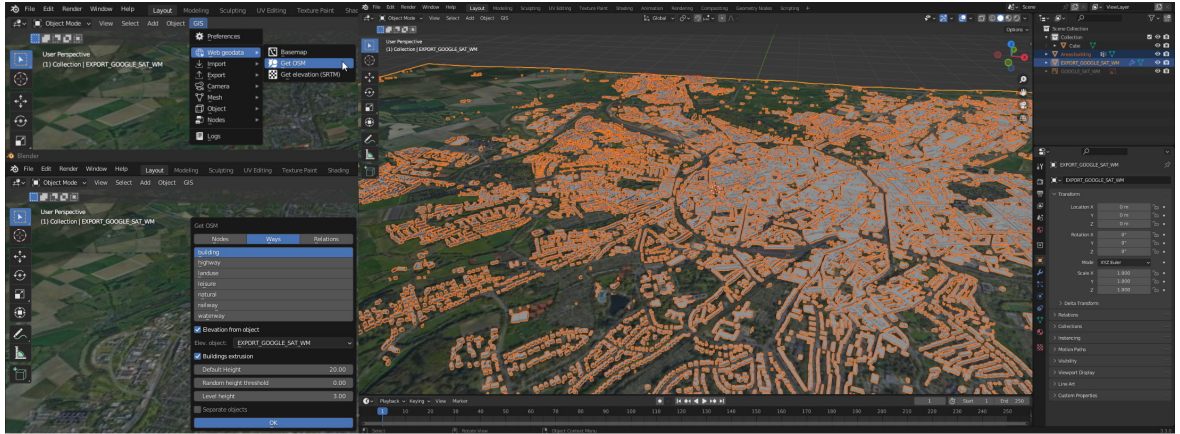


Figure 12: Modeling the Buildings using Blender GIS Plug-in.

The resulting Blender file can be saved and imported into Unity since Unity supports “.blender” files.

The next step would be to position the “XR Origin” GameObject, and with it, the “Main Camera”, according to the device’s GPS position inside the imported 3D model of Aachen. However, using the Geospatial API inside AR Foundation requires the AR-Core Extension, which was not available for AR Foundation v.5.0.2, which is required to access the 65.535m depth estimation during the thesis. Since this shall only serve as a proof of concept, the position of the “XR Origin” will be set manually without the use of GPS sensors. For more information on that, see Section 3.1.4.

In order to access the depth of the 3D model, a new camera must be added as a child to the “Main Camera” so that it follows each movement and rotation of the “Main

Camera”. This new camera will be deactivated such that it only renders its screen content if told to. Until this moment, the DEM must also be invisible to the user since it would occlude most of the screen and is only needed to extract the depth. When the DEM depth shall be extracted, the DEM must be made visible again. After that, the depth is rendered to the screen such that each pixel contains the depth of the nearest DEM fragment as $\frac{1}{distance}$, using the added camera and the implemented RenderGeometryDepth shader (see Shader 4). Using the ReadPixels function, the depth can be read from the screen [43], which takes a few milliseconds, after which the normal screen image is rendered to the screen again.

Unfortunately, this process of obtaining the DEM’s depth interferes with the user’s experience since, for a brief moment (1 of 30 frames), the screen becomes nearly entirely black, showing the depth values of the DEM. Another way of accessing the depth buffer outside of shader programs could not be found and since shader programs can only change the rendered image, rendering the depth values to the screen is the only way of accessing the depth buffer. This raises the question if it would be better to obtain the DEM’s depth on the server side if it is used only in combination with a server-side implementation of Monocular Depth Estimation.

Challenges

Using SRTM and modeling the buildings on top of the received elevation data is problematic since SRTM is a DSM (digital surface model). Therefore, the SRTM elevation data already includes the height of buildings on which the buildings are modeled additionally. Because of this, the resulting model underestimates the height difference between city buildings and nearby hills. Furthermore, the height of each building is only an estimation, and no roofs are modeled, which is why the model also underestimates the height of most buildings. Moreover, some buildings, like the RWTH Hauptgebäude, are not modeled at all. All those problems can be fixed or improved by using better DEMs.

Additionally, it must be kept in mind that the usefulness of this approach is heavily dependent on the precise positioning of the mobile, for example, through the Geospatial API of the ARCore Extension, which is not tested in this thesis.

Furthermore, rendering the DEM’s depth values to the screen is also problematic. Sampling the depth texture the same way as every other texture, as done below, returns a float of lower precision than needed.

```
float depth = texture(_CameraDepthTexture, textureCoordQuad).x;
```

Because of this, the `SAMPLE_DEPTH_TEXTURE` macro was implemented by Unity for HLSL shader programs, fixing the low precision issue. However, this macro has not been implemented by Unity for GLSL programs, which are used in OpenGL ES 3.0. Therefore, since OpenGL ES 3.0 is required for this Unity AR Foundation application, the depth can only be extracted for a certain range (see Figure 13).

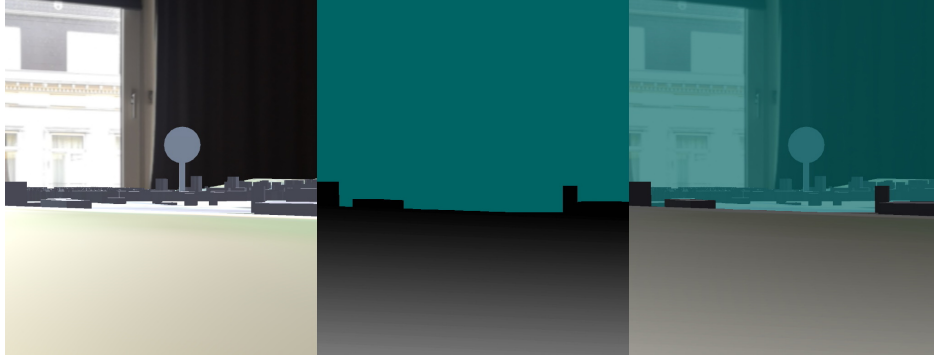


Figure 13: Bad floating point precision when sampling the depth in GLSL programs. Left Virtual content. Middle resulting DEM depth data with cyan marking the region where the depth is 0 (depth was too big). Right overlap of both images.

3.1.3 Wind Turbine Localisation in the Rendered Image

In order to locate the wind turbine in the image plane, projection geometry is used. The camera intrinsic, such as the horizontal FOV and focal length, and extrinsic are given, as well as the two points of the wind turbine (see Figure 9) in the world coordinate system.

Given a point $\mathbf{p} \in \mathbb{R}^3$ in world space coordinates, the point is first transformed into camera space coordinates, using the rotation matrix $\mathbf{R} = R_z(\gamma) \cdot R_y(\beta) \cdot R_x(\alpha) \in \mathbb{R}^{3 \times 3}$, with $R_n(\alpha)$ representing the rotation of the camera around axis n by angle α , and the translation vector $\mathbf{t} = -\mathbf{p}_c \in \mathbb{R}^3$ with $\mathbf{p}_c \in \mathbb{R}^3$ being the position of the camera in world space coordinates (Equation 2) [38].

After that, the image plane coordinates $x_p, y_p \in \mathbb{R}$ can be computed using the pinhole camera model and with the given focal length f of the camera (Equation 3) [38].

$$\mathbf{p}' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & \mathbf{1} \end{bmatrix} \cdot \begin{pmatrix} \mathbf{p} \\ 1 \end{pmatrix} \quad (2) \quad \begin{pmatrix} x_p \\ y_p \\ f \end{pmatrix} = \frac{f}{z'} \cdot \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \quad (3)$$

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

In Unity, the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$ are given by the “worldTo-CameraMatrix”, which is provided by every Unity camera [43].

The resulting point $\mathbf{p}_i = (x_p \ y_p)^T$ is in image plane coordinates with the image plane coordinate system having its origin in the middle of the image plane (see Figure 14). However, to find out whether the point will lie inside the rendered image, the size of the viewport must be computed, which is a rectangular region of the image plane that is captured by the camera given by the FOVs and the focal length of the camera (see

Equation 5 and 6). Since the point $\mathbf{p}_i$ is given in image plane coordinates having its origin in the center of the viewport, the point lies inside the viewport (rendered image) if $x_p \in [-\frac{width}{2}, \frac{width}{2}]$ and $y_p \in [-\frac{height}{2}, \frac{height}{2}]$. For a geometric interpretation, see Figure 14.

$$FOV_{vert} = FOV_{hor} \cdot \frac{Aspectratio_{vert}}{Aspectratio_{hor}} \quad (4)$$

$$height = \tan(FOV_{vert}) \cdot f \quad (5) \quad width = \tan(FOV_{hor}) \cdot f \quad (6)$$

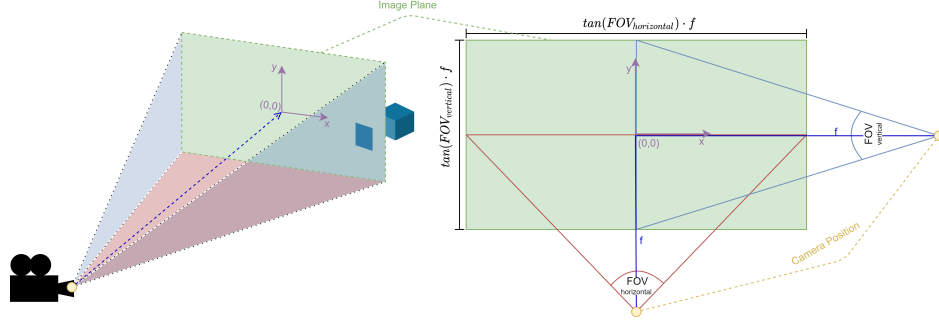


Figure 14: The left image visualizes the camera’s viewport and how an object is projected onto the image plane. The right image visualizes how the size of the viewport can be calculated using the FOVs and focal length. It should be noted that although there are two camera positions visualized in the right image, it actually represents the same camera position, but for better visibility, it has been split up.

At last, $\mathbf{p}_i$ is converted into a normalized coordinate system such that the origin of the coordinate system lies at the bottom left corner of the viewport and such that the top right corner has coordinates $(1 \ 1)^T$, as shown in Figure 15. This way, the points can be located in the rendered image regardless of its resolution.

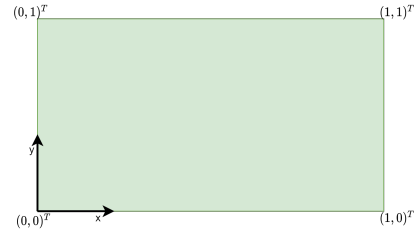


Figure 15: Visualization of the used normalized coordinate system.

Challenges

The usage of the “worldToCameraMatrix” inside the Unity Editor resulted in correct localizations of points in the rendered image. However, for unresolved reasons, using the matrix on the smartphone in combination with AR Foundation resulted in localizations that were slightly off (see Figure 16). Speculating, this may be due to camera parameters that are not correctly set within AR Foundation or due to inaccuracies in the internal updates to the “worldToCameraMatrix”. As the localization was deemed sufficiently accurate for the intended use case, the reasons were not further investigated.

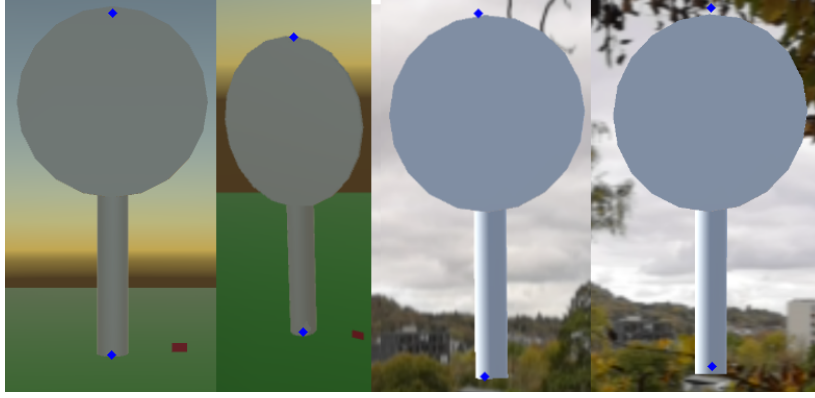


Figure 16: Visualization of the point localization inside the Unity Editor (left) and on the smartphone (right).

3.1.4 Utility Functionalities

As explained in 3.1.2, the prototype will not position the user according to GPS sensors since it shall only serve as a proof of concept. Instead, the user's position and rotation are initialized manually to capture data from different positions inside Aachen. This is done using the GUI seen in Figure 17. Apart from that, the application enables the user to visualize the depth provided by the “AR Occlusion Manager” on the screen for testing purposes (see Shaders 5 and 6). It is also possible to move the wind turbine around to make adjustments to the scene. While the motion controls in the control panel are mainly used to make slight adjustments to the active scene, the Change Scene button switches between scenes, each placing the user in another location in Aachen (Paulusstraße, Königshügel, Informatikzentrum, Lousberg).

At last, the GUI enables the user to capture all necessary data of one frame and store it such that it can be used outside of Unity (see Section 3.1.5).

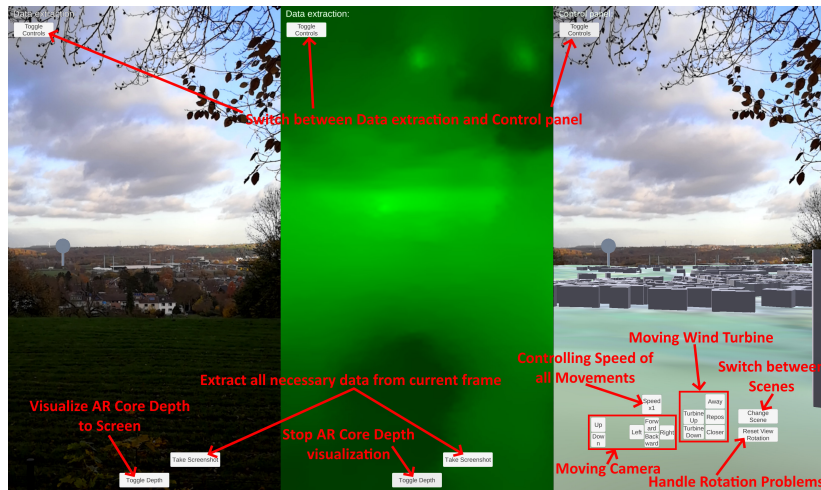


Figure 17: Visualization of the Unity Application. The left view shows the data extraction panel. The middle view shows the data extraction panel with AR Core Depth visualized on the screen. The right view shows the control panel with which the camera and wind turbine position can be changed.

3.1.5 Data Extraction

Since the Unity application is only one part of the prototype, the necessary data must be available in the python environment. In this thesis, the Unity prototype is used to create a dataset (see Section 4.1), which is used to analyze the different approaches in Section 4. This is done by pressing the “Take Screenshot” button (see Figure 17), which stores all data from the current frame to the device. This data includes the screen image, containing; the camera image occluded by virtual objects; the camera image, containing no virtual content; the DEM depth data, the dense SfM depth data, obtained through “ARCore Occlusion Manager”; the projection points; and the wind turbine distance.

However, in a final wind farm visualization application, data extraction would only be necessary if the MDE approach would be implemented on a server. Then the camera image, the DEM depth data, the projection points, and the wind turbine distance would need to be transmitted using TCP/UDP.

Overall, the data extraction consists of two steps: Data Acquisition and Data Transmission/Saving. When obtaining the data, everything must be temporarily stored within a single frame in order to avoid inconsistencies between the data. Therefore it is crucial to acquire and temporarily store the data quickly. When saving/transmitting the data, the process can be split over indefinite frames using a Unity Coroutine [43].

Data Acquisition:

To obtain the screen image, the currently active render texture must be transferred from the GPU to the CPU. This can be done by reading each pixel from the screen using the ReadPixels function [43], which is a very slow process depending on the screen resolution (here, Honor 10 with a screen resolution of 1080×2280).

The camera image must be acquired through the “ARCore Camera Manager”, returning the camera image in a previously defined resolution (in this case, the best resolution which could be selected was 1440×1080) [41]. This process is also relatively slow but can be divided into two intermediary steps, which can be split over two frames.

Acquiring the DEM depth data takes even longer than obtaining the screen image since the camera needs to actively render the depth to the screen, and then the depth can be read from the screen using ReadPixels [43]. Since the pixel values on the screen correspond to $1/DEM_distance$, it is important to store the data in a texture with either Textureformat.RFloat or Textureformat.RHalf. A texture with those texture formats provides only the red channel, and each value is a 32-bit floating point number (RFloat) or a 16-bit floating point number (RHalf) [43].

The SfM depth data can be acquired through the “ARCore Occlusion Manager” [41] and should be stored as well in a texture with either Textureformat.RFloat or Textureformat.RHalf. Since this depth data has a resolution of 120×160 , the acquisition is very fast.

The time needed to obtain the projected points and wind turbine distance is nearly

negligible.

Data Transmission/Saving:

After everything has been acquired, the data can be saved to the device or transmitted using TCP/UDP. Here, the content was saved to the device, converting the screen and camera image into PNG format while storing the environment and DEM depth data as raw bytes and writing the projected points and wind turbine distance into a txt file. Figure 37, in the appendix, displays the respective time needed for each acquisition and saving step. It stands out that saving the screen and camera image takes extremely long, while saving the DEM depth data does not. This difference occurs because the conversion into PNG format takes a very long time. Therefore, the conversion into PNG format should not be used in the final wind farm visualization application. Instead, the raw bytes should be saved/transmitted.

Challenges

The biggest challenge is the acquisition of the DEM depth data. In Unity, the depth buffer can only be accessed within shader programs, which can only modify the way and what is rendered to the screen. Therefore, rendering the depth to the screen and reading the pixels was the only solution found to access the depth buffer. Finding a better way to access the depth buffer or speeding up the acquisition process is crucial for a smooth AR experience.

3.2 Python | Monocular Depth Estimation, Sky Segmentation, Occlusion, and Analysis

3.2.1 Data Preparation

Given the dataset (see Section 2) obtained from the Unity application, as discussed in Section 3.1.5, Python is used to implement the Monocular Depth Estimation (MDE) approach from Section 2.1.2 (see Section 3.2.2) and a Sky-Segmentation-Algorithm (see Section 3.2.3). Furthermore, Python is used for an explorative analysis of the implemented approaches (see Section 4).

3.2.2 Monocular Depth Estimation (MDE)

As discussed in 2.3 and visualized in Figures 8 and 18, the images are rotated according to the wind turbine direction, which is computed using the wind turbine projection points obtained as described in Section 3.1.3. After that, the horizon will be horizontal, and the wind turbine is facing upwards.

After that, a region around the wind turbine is extracted (Cutout region), which will be fed into a neural network for depth estimation. This is done to avoid bad depth estimations as a consequence of downsampling, as described in Section 2.3.

The extracted region is then resized to the neural network's input dimensions and can then be fed into the neural network. The output is again resized to the size of the

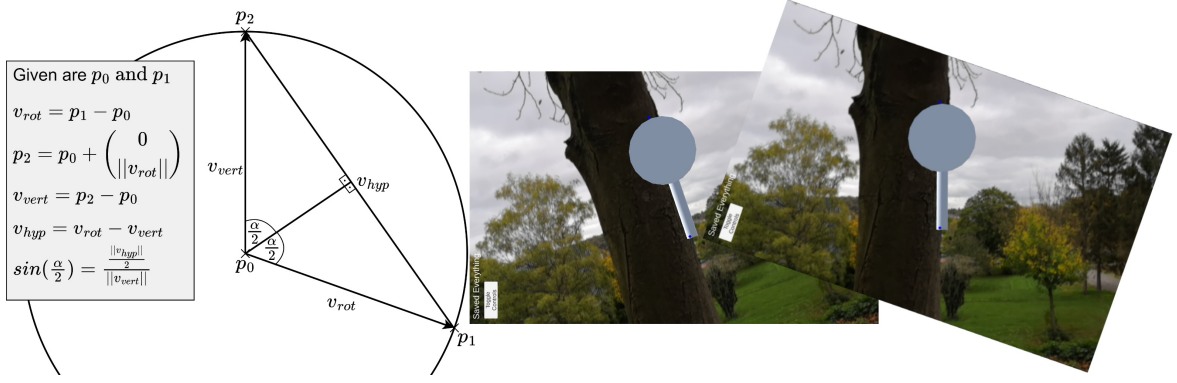


Figure 18: Rotation of the images according to the direction of the wind turbine.

extracted region.

This thesis deploys a pre-trained deep neural network developed by Alhashim and Wonka in 2018 [2] using an encoder-decoder network. To encode the RGB image into a feature vector, the DenseNet-169 network [22] was used as encoder, which was pre-trained on ImageNet [7]. The decoder consisted of a successive series of up-sampling layers and associated skip-connections. The network was chosen because it promised detailed high-resolution depth maps and qualitative better results, capturing object boundaries more carefully while having fewer parameters than state-of-the-art networks [2]. Alhashim and Wonka trained the network using the NYU Depth v2 [37] dataset, consisting of indoor scenes with an upper depth limit of 10 meters, and the KITTI [16] dataset, consisting of urban scenes with an upper depth limit of 80 meters.

3.2.3 Sky-Segmentation-Algorithm

The Evaluation (see Section 4.3 and 4.6) revealed that both, SfM and MDE approaches suffer in particular from the misclassification of sky regions. As compensation, the following sky segmentation algorithm has been developed. Developed toward the end of the thesis project, it was kept fairly simple. Nonetheless, the suggested approach turned out as an efficient algorithm with good performance, as discussed in Section 4.4.

The utilized approach is a simplified version of K-Means [13, 26]. Generally, K-Means is a clustering algorithm that can be used for image segmentation as clustering and is guaranteed to converge after a finite amount of update steps to a local minimum. (1) At the start, K-Means randomly initializes K clusters inside the selected feature space. (2) Then, each pixel is assigned to its nearest cluster. (3) After that, each cluster is updated using the mean of all pixels that were assigned to the cluster. Steps 2 and 3 are repeated until the clusters converge.

Normally, it is not simple to choose the optimal number of clusters K. However, in this case, there are two natural classes, sky, and non-sky. Therefore, K was chosen to be 2. Moreover, the random initialization and iterative process can be avoided since

the cluster centers can be estimated by selecting a region mainly containing non-sky pixels and a region mainly containing sky pixels. For non-sky region selection, the DEM depth data can be used. For sky region selection, the wind turbine direction can be followed, as seen in Figure 19, until the border of the image is reached. After that, both clusters can be initialized by computing the mean of the non-sky and sky regions, after which assigning each pixel to its nearest cluster is trivial. However, the resulting segmentation is only accurate under the assumption that the sky selection was successful.

This way, the random initialization and iterative process are not needed, speeding up the approach enormously. When using the RGB feature space and a resolution of 120×160 , the utilized algorithm delivers good results even when the selected sky region is traversed by leaves and branches (see Section 4.4). Nevertheless, an option to improve accuracy could be to communicate to the user which region is used to compute the sky cluster and should therefore contain mainly sky pixels by visualizing the region on the screen.



Figure 19: Left image: the sky region (red) is selected by following the direction of the wind turbine direction, and the ground region (yellow) is selected using the DEM depth data. Middle image: ground truth map with white marking foreground pixels and black marking background pixels. Right image: output of the sky segmentation algorithm, white marking sky pixels and black marking non-sky pixels.

3.2.4 Computing Occlusion Maps and Combining Them

The following defined occlusion maps have the same connotation as the occlusion map defined in Section 2.2:

$$Occlusion_Map(i, j) = \begin{cases} True, & \text{if pixel}(i, j) \text{ is in front of Scene-Object} \\ False, & \text{else (background unknown pixel)} \end{cases} \quad (7)$$

SfM Occlusion Map

The SfM approach for Close-DDE from Section 2.1.1, implemented as described in Section 3.1.1, yields the dense SfM depth map, which contains absolute depth values in a range of 0 to 65.535m for each pixel of the entire image. Consequently, the entire image is classified as foreground if the wind turbine distance exceeds 65.535m. To counteract, an SfM threshold is set so that pixels assigned a larger depth than this threshold are classified as background. Additionally, the SfM threshold discards low confidence depth estimations since SfM loses confidence with increasing distance [18]. A suitable threshold is derived in the evaluation of SfM in Section 4.3.1.

Given a suitable SfM threshold, the SfM occlusion map, which is evaluated in Section 4.3, is computed as follows:

$$SfM_Occlusion(i, j) = \begin{cases} True, & \text{if } SfM_depth(i, j) < SfM_threshold \\ False, & \text{else (background/unknown pixel)} \end{cases} \quad (8)$$

Sky Occlusion Map

The Sky Segmentation approach from Section 2.1.4 implemented as described in Section 3.2.3 yields the sky segmentation, which classifies the entire image into sky and non-sky pixels:

$$Sky_Segmentation(i, j) = \begin{cases} True, & \text{if pixel}(i, j) \text{ is a sky pixel} \\ False, & \text{else (foreground/unknown pixel)} \end{cases} \quad (9)$$

Since sky pixels are background pixels, the sky occlusion map, which is evaluated in Section 4.4, is derived as follows:

$$Sky_Occlusion(i, j) = \neg Sky_Segmentation(i, j) \quad (10)$$

Therefore, the sky occlusion map occludes the virtual objects by everything that is not detected to be a sky pixel.

MDE Occlusion Map

The MDE approach for Long-DDE from Section 2.1.2, implemented as described in Section 3.2.2, yields the MDE depth map, which contains relative depth values for each pixel inside the Cutout region. Unfortunately, those relative depth values cannot be compared to the wind turbine's absolute distance without knowing the scale of the scene (see Section 2.1.2). However, as the problem of inferring depth values from 2D images is ill-posed, the use of relative depth values is a standard in training neural networks for depth estimation, which experimentally leads to better performance.

As described in Section 2.3, to bypass the issue of relative depth values, the DEM depth data is used to select a region inside the MDE depth map, which should be in front of the wind turbine. Selecting this foreground region inside the MDE depth map

will be referenced in the following as Region Selection while using the DEM depth map for Region Selection will be referenced as topographic Region Selection.

After such a region has been selected, a threshold can be computed, using the containing relative MDE depth values, that classifies the MDE depth map values into foreground and background. However, due to inaccuracies inside the DEM and MDE depth data, which cannot be assessed at this stage, four Thresholding Methods (see Table 1) are introduced, which are evaluated and compared against each other in Section 4.6.2.

MAX: returns the maximum relative depth value of the selected region.

UF (Upper Fence): returns the upper fence of the selected region.

MIN(UF,MAX): returns the minimum of the MAX and UF methods.

Q3: returns the third quartile of the selected region.

Table 1: The four introduced Thresholding Methods, descending, choosing the MDE threshold more carefully.

While MAX is not able to discard outliers for threshold selection, UF can. However, UF can also exceed the MAX threshold. Because of that, MIN(UF,MAX) incorporates both MAX and UF by choosing the minimum of both. Therefore, outliers can be discarded while ensuring that the threshold does not exceed the maximum relative depth value inside the selected region. Q3 is the most careful method, able to discard outliers but also discarding way more samples than UF.

Using the Region Selection and Thresholding Methods above, the MDE threshold is derived automatically for the Cutout region, and the MDE occlusion map is computed as follows:

$$\text{MDE_Occlusion}(i, j) = \begin{cases} \text{True}, & \text{if } \text{MDE_depth}(i, j) < \text{MDE_threshold} \\ \text{False}, & \text{else (background/unknown pixel)} \end{cases} \quad (11)$$

Combining the Occlusion Maps

Using the above-defined occlusion maps, the approaches are combined as described in Section 2.3.

Since the SfM and MDE occlusion maps both suffer from the misclassification of sky regions, they are complemented using the Sky Segmentation, setting all detected sky pixels as background resulting in the SfM sky and MDE sky occlusion map, which are evaluated in Section 4.5.2, and Section 4.6.3:

$$\text{SfM_sky_occlusion}(i, j) = \text{SfM_occlusion}(i, j) \wedge \neg \text{Sky_Segmentation}(i, j) \quad (12)$$

$$\text{MDE_sky_occlusion}(i, j) = \text{MDE_occlusion}(i, j) \wedge \neg \text{Sky_Segmentation}(i, j) \quad (13)$$

Using the SfM sky and MDE sky occlusion maps, the combined occlusion map can be computed, which is evaluated in Section 4.7:

$$\text{Comb_occlusion}(i, j) = \text{SfM_sky_occlusion}(i, j) \vee \text{MDE_sky_occlusion}(i, j) \quad (14)$$

4 Evaluation

4.1 Dataset

In order to evaluate the utilized algorithms, a dataset¹¹ was created using the implemented Unity Application. The dataset is split into four sequences, each captured from different locations inside Aachen. The sequences are made up of scenes, with a single scene representing the ensemble of all available data for one frame, namely the screen and camera image, the DEM and SfM depth data, the points of the wind turbine in the image plane, and a ground truth map that was created by hand in the form of a binary label (see Figure 20).



Figure 20: Example visualization of a scene. From left to right: screen image, camera image, ground truth map (white representing foreground and black background pixel), DEM depth map, SfM depth map. Depth maps: brighter color equals greater distance; cyan marking regions with no depth.

Sequence	Scenes	Properties
Paulusstraße	7	all background pixels are sky pixel
Königshügel	13	small proportion of non-sky background pixels
Informatikzentrum	29	many leaves and branches occluding wind turbine; small proportion of non-sky background pixels
Lousberg	12	greatest distance between device and wind turbine; big proportion of non-sky background pixels

Table 2: Overview of the dataset.

The Paulusstraße Sequence was captured from an apartment on the 4th floor. This sequence is the least complex since all background pixels are sky pixels. The Königshügel Sequence was captured standing on the Königshügel and contains a small proportion of non-sky background pixels, which get consequently harder to be differentiated from foreground pixels. The Lousberg Sequence was captured standing on the Lousberg and is complex because of the great distance of the wind turbine and because it contains a big proportion of non-sky background pixels. The Informatikzentrum Sequence was captured in front of the Informatikzentrum and contains the most complex scenes. The

¹¹<https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset>

most complex images contain a lot of trees, leaves, and branches, occluding the smart-phone’s view very closely. The less complex scenes also contain leaves and branches but much fewer. The least complex scenes do not contain close leaves but walls instead.

Overall, the entire dataset consists of 61 scenes unevenly distributed among the sequences. Despite the small size of the dataset and the fact that all scenes were taken inside Aachen, care was taken to have a representative sample for typical use cases and challenges for wind farm visualization (see Figure 21).

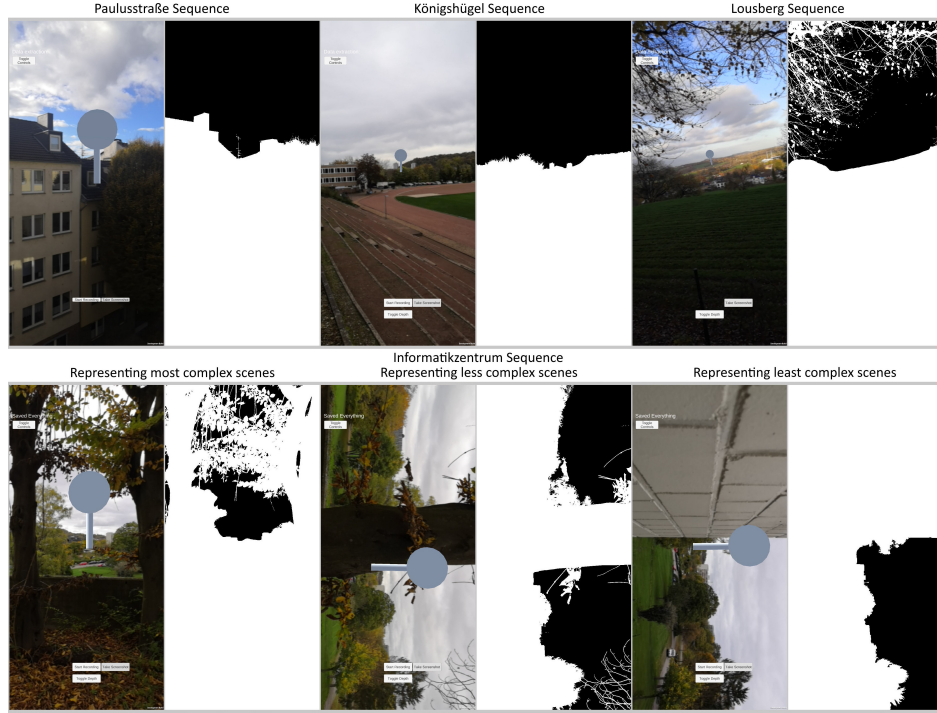


Figure 21: Visualization of the sequences: Screen images and corresponding binary ground truth maps (white: foreground pixels, black: background pixels). For the entire sequences, see Figures 39, 40, 42, and 41 in Appendix.

4.2 Metrics

For each scene, the prototype yields occlusion maps resulting from the different approaches. Each of those approaches is evaluated against a ground truth occlusion map in order to assess its contribution to the combined occlusion map, as defined in Section 3.2.4. For a numerical analysis of the occlusion maps, two metrics are used:

1. The False Positive Rate (FPR) is the rate of pixels falsely classified as foreground pixels (falsely occluding the virtual objects).

$FPR = \frac{FP}{FP+TN}$, with FP denoting the number of pixels falsely classified as foreground pixels and $FP + TN$ the number of background pixels.

2. The False Negative Rate (FNR) is the rate of pixels falsely classified as background pixels (falsely getting occluded by the virtual objects).

$FNR = \frac{FN}{FN+TP}$, with FN denoting the number of pixels falsely classified as background pixels and $FN + TP$ the number of foreground pixels.

As explained in Section 2.3, each pixel is classified in either foreground or background/unknown. Because of this, the introduced metrics focus on the foreground pixels and not on those classified correctly as background/unknown. This choice resulted from the fact that only the foreground classification influences the resulting occlusion since, by default, everything is classified as background (see Section 2.2).

In order to compute the FPR and FNR, each occlusion map is compared with the ground truth occlusion map. To visualize the above metrics, a comparison map is computed (for an example, see Figure 22).

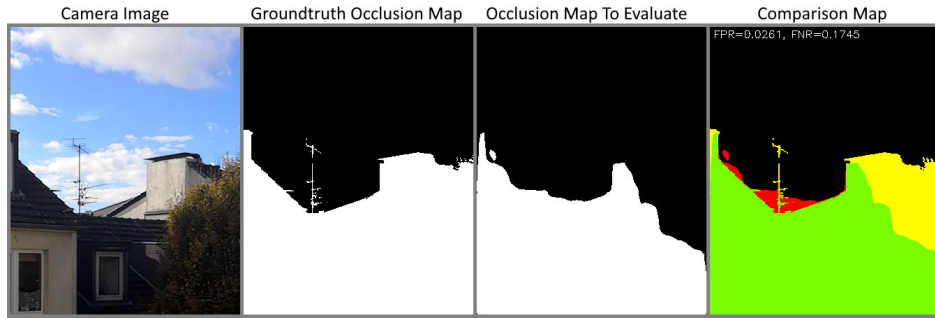


Figure 22: Derivation process of the comparison map. Last image represents the comparison of both occlusion maps. White: foreground pixels. Black: background pixels. **Green:** pixel correctly classified as a foreground pixel. **Yellow:** pixel falsely classified as a background pixel (FNR). **Red:** pixel falsely classified as a foreground pixel (FPR). Black: pixel correctly classified as a background/unknown pixel (not of interest).

4.3 Evaluation of SfM

This section evaluates the SfM approach from Section 2.1.1 implemented as described in 3.1.1 and its resulting SfM occlusion map according to Section 3.2.4.

While Section 4.3.1 explains how the SfM threshold, which is used to compute the SfM occlusion map out of the SfM depth map (see Section 3.2.4), is derived, Section 4.3.2 evaluates the chosen threshold and the resulting occlusion map.

4.3.1 Fine Tuning SfM Threshold

To find a suitable SfM threshold, the FPR and FNR are computed for each image in each sequence for every possible threshold (see Figure 24). Here, the rates were computed for thresholds from 0 to 65m with 1-meter steps. However, the FPR and FNR are inversely related, meaning that with increasing SfM threshold, the FPR increases while the FNR decreases. Therefore, they are weighted against each other.

Normally the FPR should be valued much higher than the FNR when analyzing SfM because the SfM algorithm shall only detect close foreground pixels. And since a foreground classification through SfM cannot be reversed by the Long-DDE, the FPR represents an irreversible misclassification, while the FNR represents a misclassification that can be fixed by an additional Long-DDE approach.

Unfortunately, according to the evaluation, the utilized SfM algorithm suffers especially from the misclassification of sky regions. Figure 23 shows an example where sky regions are predicted to be in front of the majority of the scene.



Figure 23: Purposive sample representing the Informatikzentrum Sequence with corresponding SfM depth maps. Dark colors represent close-depth, and bright colors represent far-depth estimations. For the entire scenes, see Figure 41 in Appendix.

Speculating this might be due to the small resolution (120×160) of the image, which is fed into the algorithm, and the fact that sky regions are mostly textureless areas. Since the algorithm is not able to estimate depth in textureless regions, the depth is inferred during the creation of the dense depth map using the surrounding estimated depth values, as explained in 2.1.1, leading to erroneous estimations.

Therefore, it is inadvisable to weigh the FPR too hard since misclassified sky regions significantly contribute to the FPR. Instead, the FPR and FNR must be weighted more carefully against each other to find a good trade-off between FPR and FNR in the context of the misclassification of sky regions and to avoid, at the same time, the favoring of a threshold resulting in no occlusion.

By computing the FPR and FNR for all possible thresholds (see Figure 24) for each scene, the median FPR and FNR for all thresholds and each sequence can be derived (see Figure 25). The median was used here since it is less sensitive to outliers than the mean.



Figure 24: SfM: Visualization of the FPR and FNR on a single scene with increasing threshold from left to right. See Figure 22 for color interpretation.

A single choice of threshold for all scenes and sequences is not optimal in all cases, in particular since large fluctuations of the FPR and FNR were observed over the

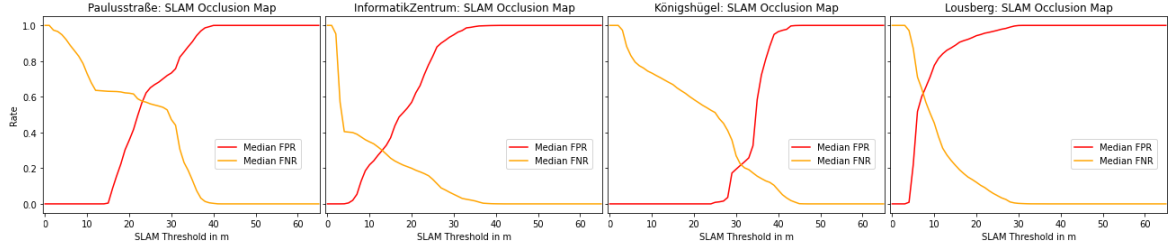


Figure 25: SfM: Plots showing the median FPR and FNR of all scenes of a sequence plotted against the independent variable SfM Threshold and grouped by the sequence membership.

different scenes and sequences. In this thesis, the global SfM threshold is set to 15m based on the numerical analysis above, practical experience with the algorithm, and visual evaluation of the yielded occlusion results.

4.3.2 SfM Occlusion with 15m SfM Threshold

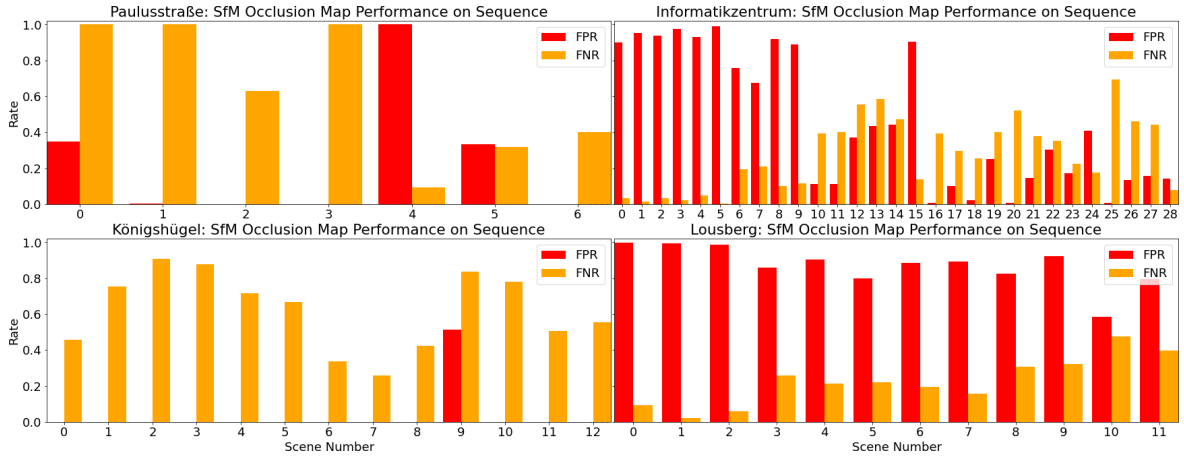


Figure 26: SfM: FPR and FNR for each image in all four sequences for a chosen threshold of 15m.

For a fixed SfM threshold, the evaluation is straightforward. The plots in Figure 26 show the FPR and FNR for each scene in each sequence. It stands out that the first scenes of the Informatikzentrum Sequence have high FPR, which decreases over the sequence. This is due to the fact that the most complex scenes are located at the beginning of the sequence, and the sequence gets simpler towards the end. Furthermore, the entire Lousberg Sequence performs poorly with the 15m SfM threshold.

The main source of error using the SfM algorithm for depth estimation are textureless and distant regions where depth data can not be inferred. Therefore, the SfM yields a sparse depth map, which is then used to create a dense depth map by inferring the missing depth values from the depth of neighboring pixels. The estimated depth of distant regions (e.g., sky regions) is, therefore, highly dependent on the depth of surrounding pixels.

In the Informatikzentrum and Lousberg Sequence, most sky regions are traversed by leaves and branches, which are estimated to have a small depth. Therefore, the depth values of the leaves and branches are used to infer the depth values of the adjacent sky regions resulting in poor performance.

Finally, an aspect of the evaluation not covered by the numerical evaluation above is the following: The SfM algorithm applies temporal smoothing. Hence, the SfM depth is not only dependent on the comparison between current and previous camera images but also on the previous depth estimations. While temporal smoothing can be a significant source of error in single image evaluation, it can significantly improve user experience inside the Unity app. This can be frequently experienced inside the Unity Application.

4.4 Evaluation of Sky Segmentation for Occlusion

This section evaluates the Sky-Segmentation-Algorithm approach from Section 2.1.4 implemented as described in 3.2.3 and its resulting sky occlusion map according to Section 3.2.4.

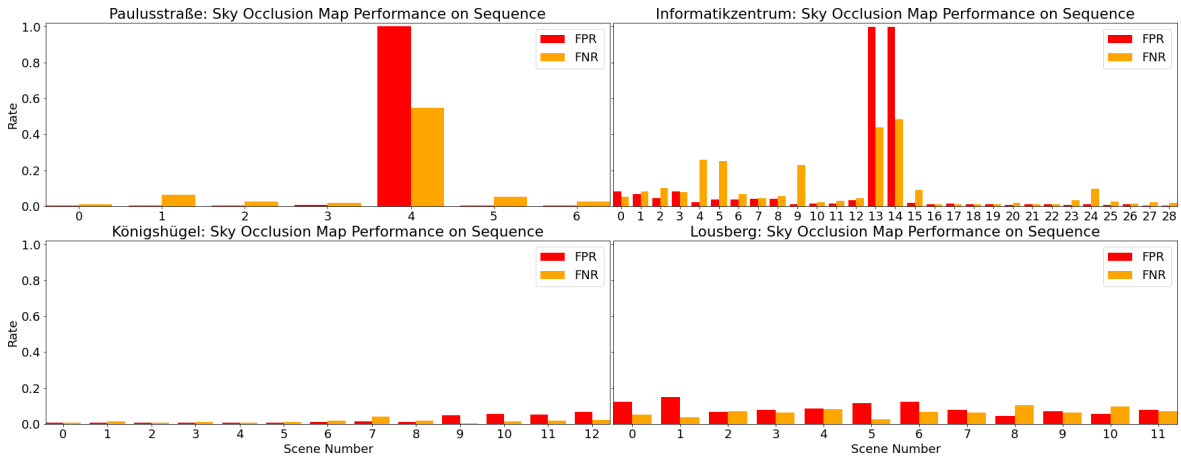
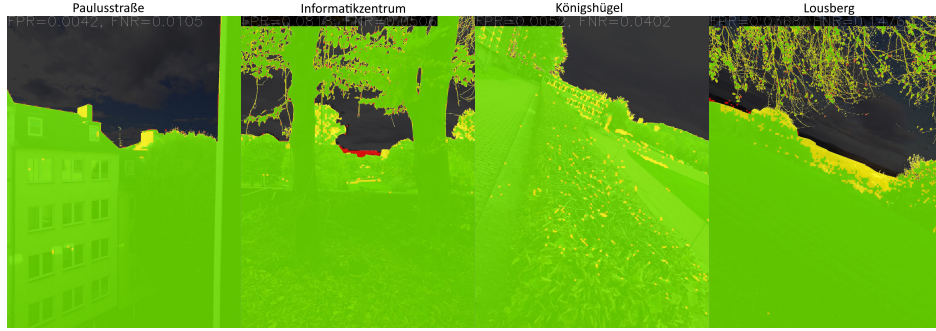


Figure 27: Sky: FPRs and FNRs plotted for each image in each sequence. FPR and FNR are typically very small in comparison to the other methods. The peaks originate from the fact that the region selection went wrong.

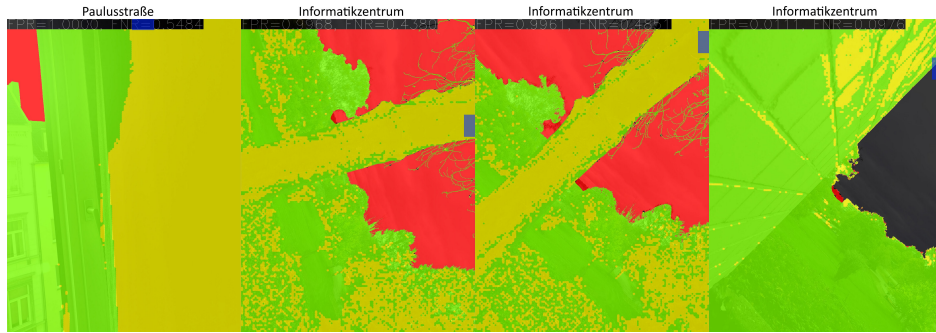
Despite the simplicity of the applied algorithm, the yielded results outperform the SfM occlusion with a 15m SfM threshold by far (see Figure 27). With a few exceptions, the algorithm outperforms the SfM approach on each scene (compare Figure 27 with Figure 26). In total, when comparing the SfM approach with a 15m SfM threshold against the Sky-Segmentation-Algorithm, the mean FPR of all Scenes is reduced by 34.47% from 42.27% to 7.80% while the total mean FNR is reduced by 30.49% from 38.06% to 7.57% (see Table 3).

The noticeable peaks inside the Paulusstraße and Informatikzentrum sequences in Figure 27 are attributable to wrong sky cluster initialization (see Figure 28b) being

the consequence of the assumption that a region, mainly containing sky pixels, can be found when following the direction of the wind turbine towards the image border (see Section 3.2.3).



(a) Successful occlusion results obtained through Sky Segmentation for Occlusion.



(b) Most problematic results obtained through Sky Segmentation for Occlusion. Blue pixels mark the regions used to compute the sky cluster center.

Figure 28: Sky: Visualization of the occlusion results obtained through Sky Segmentation. Here, the comparison maps were laid over the camera image in order to provide a better understanding of the scene. See Figure 22 for color coding explanation.

Besides the overall good occlusion results, the implemented Sky-Segmentation-Algorithm is computationally efficient. When executing the algorithm on an AMD Ryzen 5 2500U, a middle-class laptop CPU from 2018, the segmentation of a 120×160 resolution image takes around 0.015s with peaks of 0.0065s and 0.031s using a simple python implementation, including the computation of the confidence with which the algorithm is able to classify a pixel as sky pixel (see Figure 38).

The performance with regard to efficiency and accuracy confirms that this algorithm may be a good choice for implementation on the smartphone itself. In order to ensure a smooth augmented reality experience at 30fps, each frame must be processed within 0.033s, which the algorithm accomplishes. A C# implementation may even outperform the current python version in speed.

However, the algorithm suffers from major limitations, such as its inability to classify non-sky pixels as background pixels (see red regions in Figure 28). As discussed

in Section 2.2, Sky Segmentation can only solve two-thirds of the problem. Therefore, it must be combined with another depth estimation technique. Additionally, the simplified k-means algorithm only considers the color distribution of the pixels, not their spatial information, resulting in a grainy output, as seen in the middle right image of Figure 28a.

4.5 Evaluation of SfM + Sky Occlusion

This section evaluates the combination of the SfM and Sky-Segmentation-Algorithm approach and its resulting SfM sky occlusion map according to Section 3.2.4. During the evaluation, a new SfM threshold is derived in Section 4.5.1 and evaluated in Section 4.5.2.

4.5.1 Fine Tuning SfM Threshold

By combining SfM with Sky Segmentation, the main source of error of SfM, misclassification of sky regions, is fixed, resulting in different error rates among the sequences. While the previously derived SfM threshold compromised the misclassification of sky regions with the quality of the yielded occlusion results, this is not necessary anymore. Computing the median FPR and FNR for all sequences (see Figure 29) shows the potential of a higher SfM threshold due to extremely low FPR for all possible SfM thresholds.

However, since sky background pixels cannot be misclassified anymore, thanks to the Sky-Segmentation-Algorithm, only non-sky background pixels (pixels that should be behind the wind turbine but are not detected by the Sky-Segmentation-Algorithm) contribute to the FPR. Additionally, the proportion of non-sky background pixels compared to sky background pixels is extremely small inside the images. Therefore, the FPR will remain very small, even if all non-sky background pixels are misclassified as foreground. However, correctly classifying the non-sky background pixel is extremely important to yield good occlusion results since, otherwise, the wind turbine would be occluded by non-sky objects that are actually kilometers behind the wind turbine. In order to represent the proportion of misclassified non-sky background pixels, the FPR of non-sky background pixels is computed and plotted (cyan) in Figure 29. When choosing the SfM threshold, the focus should be on keeping the FNR as well as the non-sky background FPR low.

After evaluating the non-sky background FPR and FNR against each other while also taking into account practical experience with the algorithm and yielded occlusion results, a threshold of 30m is chosen. However, the non-sky background FPR and the FNR are still subject to strong fluctuations among the different scenes, and the chosen threshold is not optimal for every scene or sequence.

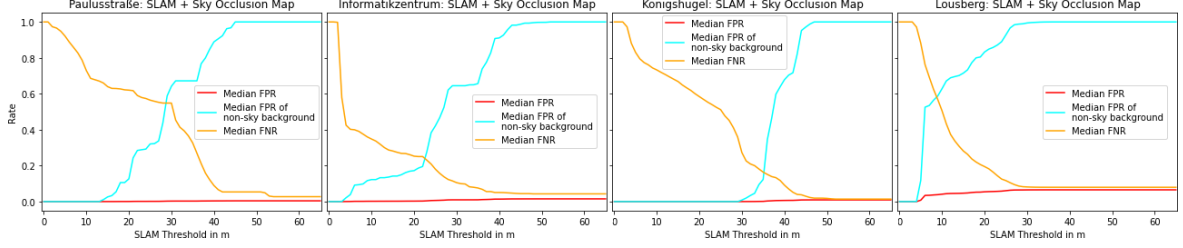


Figure 29: SfM+Sky: Plots showing the median FPR and FNR plotted against the independent variable SfM Threshold and a re-scaled version of the median FPR, representing only the non-sky background pixels FPR under the assumption of accurate Sky Segmentation.

4.5.2 SfM 30m + Sky Occlusion

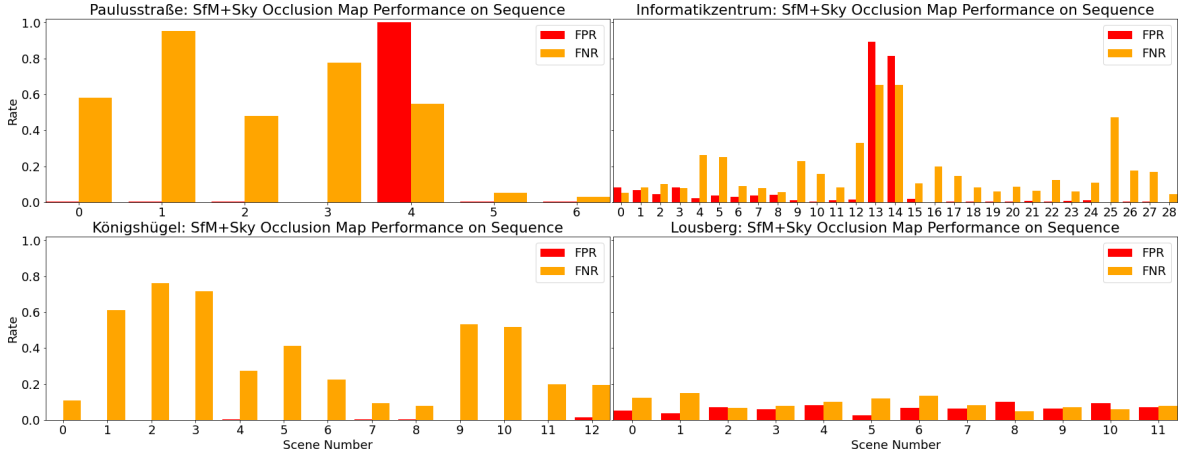


Figure 30: SfM+Sky: Mean FPRs and FNRs plotted for each scene in each sequence.

As seen during threshold selection, by correcting SfM with Sky Segmentation, a much higher SfM threshold can be chosen, reducing the overall FNR, while the Sky Segmentation keeps the FPR low (see Figure 30). Looking at the improvement from SfM with a 15m SfM threshold to SfM+Sky and 30m SfM threshold, the mean FPR of all scenes is reduced by 35.61% from 42.27% to 6.66% because of the Sky Segmentation, while the mean FNR of all scenes is reduced by 14.64% from 38.06% to 23.42% through the increased threshold (see Table 3).

Despite significant improvement, a few scenes (Paulusstraße scene 4 and Informatikzentrum scenes 13 and 14) have deteriorated because of erroneous Sky Segmentation. Because of the way, the SfM occlusion map is complemented with the Sky Segmentation (see Section 3.2.4), erroneous sky segmentations cannot be corrected by SfM and will persist.

4.6 Evaluation of MDE

This section evaluates the MDE approach from Section 2.1.2 implemented as described in 3.2.2 and its resulting MDE occlusion map according to Section 3.2.4, as well as the

combination of MDE and Sky Segmentation and their resulting MDE sky occlusion map according to Section 3.2.4. While Section 4.6.1 highlights the improvement by combining MDE with Sky Segmentation, Section 4.6.2 evaluates the DEM for Depth Estimation approach from Section 2.1.3 implemented as described in Section 3.1.2 for topographic Region Selection as described in Section 3.2.4, and shows how a Thresholding Method is selected, which will automatically determine the MDE threshold for each MDE depth map as described in Section 3.2.4.

While previous evaluations considered the entire image when computing the FPR and FNR, the following evaluations will be limited to the Cutout region (as described in Section 2.3) as this is the only area of the image that is fed into the neural network and for which a depth map and occlusion map are generated (see Section 2.3). Therefore, the following FPR and FNR values are not comparable with the previously derived error rates since they are computed over different regions.

During the SfM evaluation, the focus was on keeping the FPR low while minimizing the FNR as a secondary goal because of the range limitation of SfM and the decreasing confidence with growing distance. This was feasible under the assumption that the Long-DDE would complement important missing foreground classifications. In other words, the SfM approach should only detect close foreground pixels inside the image, while the MDE should detect all foreground pixels inside the Cutout region. Nevertheless, it is still better to avoid falsely occluding the virtual object than falsely not occluding it, as described in Section 2.2. Hence, keeping the FPR low should be valued a little bit more important than keeping the FNR low when weighting the error rates against each other.

4.6.1 MDE vs. MDE + Sky

As seen in Section 4.5, combining SfM and Sky Segmentation significantly improves the SfM occlusion maps. Figure 31 shows that the combination of MDE and Sky Segmentation, as described in Section 3.2.4, significantly decreases the FPR but also increases the FNR for all combinations of Region Selection and Thresholding Methods.

Two major reasons could be found for the worsening of the FNR. First, since the Sky Segmentation has a very small resolution and must therefore be up-sampled during the occlusion process, areas finely traversed by leaves and branches may not be classified precisely for every pixel. The second reason is that distant objects get a hazy outline because of Aerial Perspective (see Section 2.1.2), which makes the Sky Segmentation prone to misclassifying those regions as sky regions. However, if the MDE would be applied on a server and not on a smartphone, the Sky-Segmentation-Algorithm could be implemented computationally more expensively. Using a higher resolution or making the Sky-Segmentation-Algorithm more robust to hazy outlined regions may decrease the FNR.

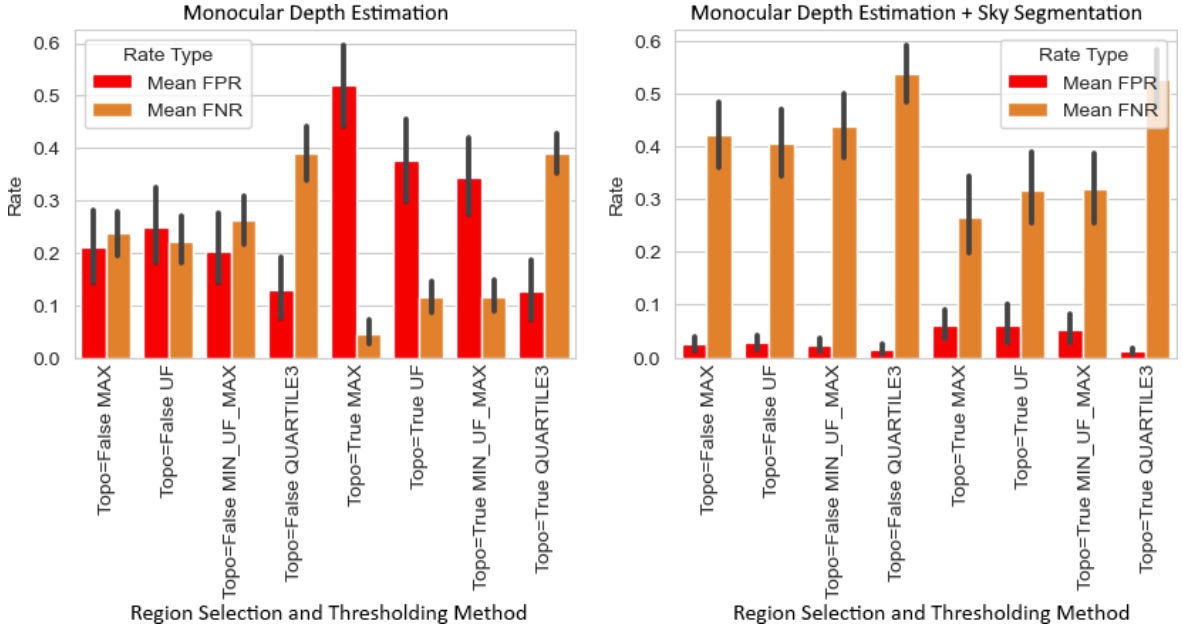


Figure 31: MDE: Mean FPR and FNR of all sequences plotted for each combination of Region Selection and Thresholding mode. The left plot shows the Mean FPR and FNR of the resulting occlusion map when solely using MDE, and the right plot corresponds to the resulting occlusion map when correcting the MDE with the Sky Segmentation.

Nevertheless, regardless of the increased FNR, complementing the MDE occlusion map with the Sky Segmentation results in visually better occlusions. Furthermore, the FPR approximately decreases by a factor of 10, while the FNR approximately increases by a factor of 2. Because of that and the fact that the FPR is still valued as slightly more important, the following evaluations will focus on the combined approach of MDE and Sky Segmentation.

4.6.2 Fine Tuning Region Selection and Thresholding Method for MDE + Sky

This section compares the different combinations of Region Selection with Thresholding Methods, which are used to create the MDE occlusion map, and evaluates the occlusion results when complementing the resulting MDE occlusion map with the Sky Segmentation as described in Section 3.2.4.

In order to prove the concept of topographic Region Selection (also referenced as Topo=True), as described in Section 3.2.4, it is compared for all Thresholding Methods against a simple Region Selection, which simply selects a region inside the depth map below the wind turbine (also referenced as Topo=False).

Furthermore, the different Thresholding Methods, as defined in Table 1, are compared against each other. In addition to the previously defined Thresholding Methods, a threshold is chosen by hand as a reference value to visualize how a human would have chosen the threshold to yield the best possible occlusion using the already computed

MDE depth map. The derived MDE thresholds are then used to compute the MDE occlusion map (see Section 3.2.4), which is complemented using Sky Segmentation, yielding the MDE sky occlusion map (see Section 3.2.4). This combination is then compared against the ground truth map (see Figure 32).

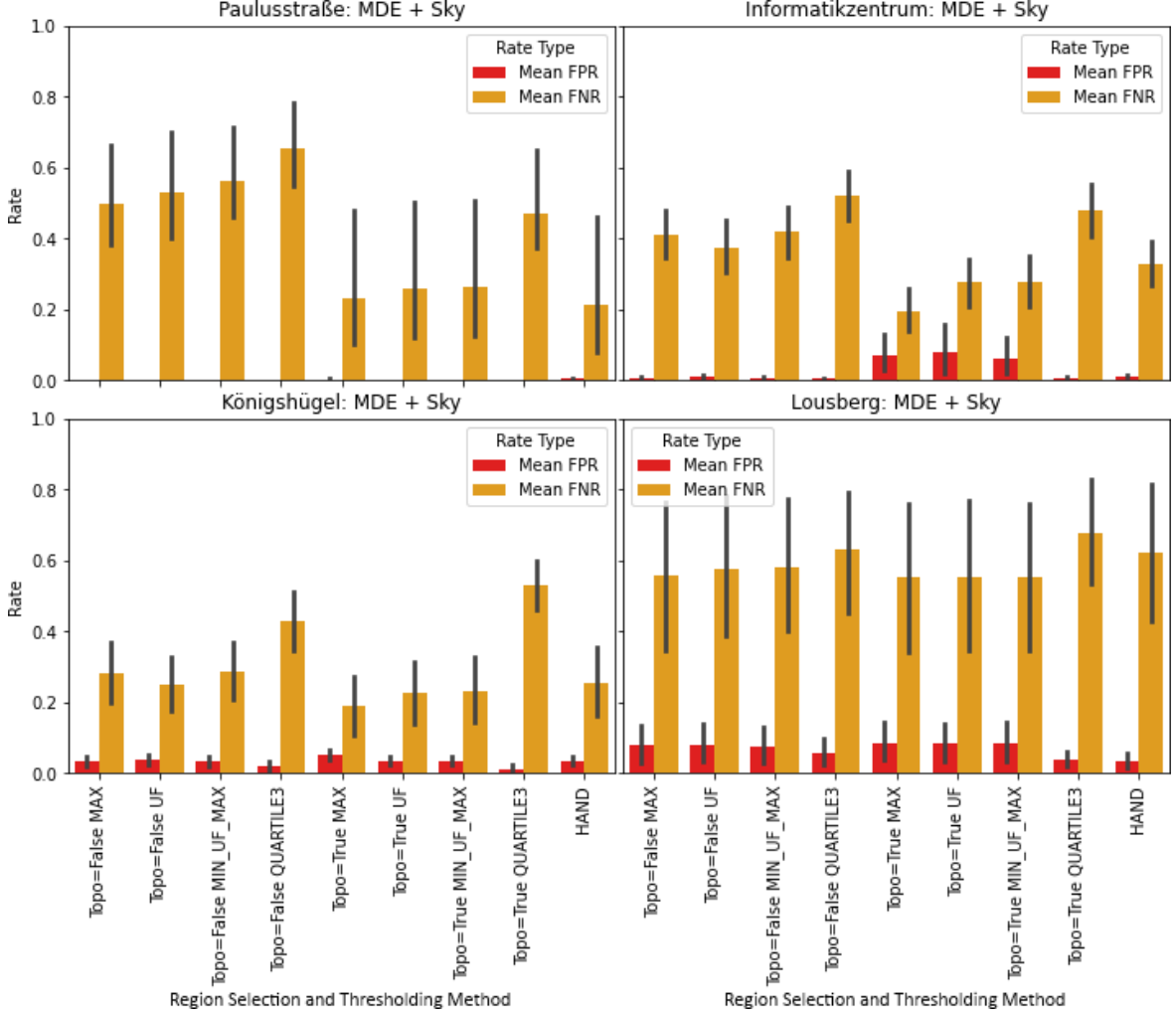


Figure 32: MDE+Sky: Mean FPR and FNR for each sequence plotted for each combination of Region Selection and Thresholding Method. While Topo=False states that the simple Region Selection method was used, Topo=True states that the topographic Region Selection was used.

Focusing on the evaluation of topographic Region Selection, Figure 32 shows that topographic Region Selection overall decreases the FNR significantly while also increasing the FPR significantly. However, the decrease in FNR is significantly greater than the increase in the FPR, and the hand-derived reference threshold is modeled more closely when using topographic Region Selection. Therefore, the yielded occlusion results significantly improve when using topographic Region Selection, which could also be verified by visually evaluating the occlusion results. Hence, topographic Region

Selection is chosen over the simple Region Selection method.

Focusing on the evaluation of the different Thresholding Methods while using topographic Region Selection, Figure 32 shows that the error rates of the reference threshold (chosen by hand) are very similar to the error rates of MAX, UF, and MIN(UF,MAX). However, it is not immediately discernible from the plot which Thresholding Method performs best when compared to the reference threshold. Therefore, the mean squared error (MSE) has been computed for each Thresholding Method by comparing the automatically chosen thresholds with the reference threshold resulting in the following mean squared errors: MAX 0.42334; UF 0.28746; MIN(UF,MAX) 0.28087; Q3 0.31672. Under the assumption that the reference threshold is the best possible threshold, the mean squared error shows that the MIN(UF,MAX) Thresholding Method performs best.

4.6.3 Evaluation MDE+Sky with Topographic Region Selection and MIN(UF,MAX)

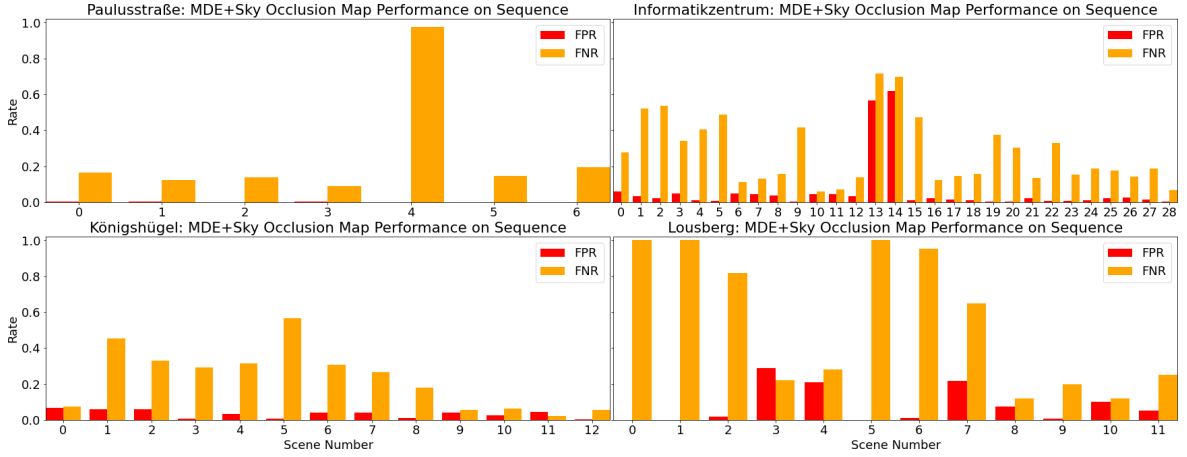


Figure 33: MDE+Sky: FPRs and FNRs plotted for each scene in each sequence.

After topographic Region Selection and MIN(UF,MAX) have been shown to yield the overall best results, the FPR and FNR of the Cutout can be plotted for each scene (see Figure 33). However, when comparing this plot with previously derived FPR and FNR plots, it should be considered that the previous plots during SfM and Sky evaluation were computed over the entire image, while the FPR and FNR in the evaluation of MDE are only plotted over the Cutout, a smaller part of the images.

In total, most of the results look promising but not outstanding. Speculating, this might be attributable to erroneous depth estimations from the neural network. Figure 34 shows evidence that the neural network is prone to a specific depth estimation pattern. The same pattern was observed in the MDE depth maps, leading to mixed occlusion results, especially at the borders of the Cutout region. This pattern can also be found in the NYU Depth v2 [37] and KITTI [16] datasets, on which the neural network was trained. The NYU Depth v2 [37] dataset consists of indoor images, which

overall have a wall to the left and/or right, and the bottom and top of the images are occupied by the ground or ceiling. The KITTI [16] dataset consists of outdoor images taken from the roof of a car. Therefore, most images have buildings or parking cars to the left and right while the street is expanding towards the center of the image.

This overfitting to the pattern is suspected to be the reason for the mixed results. Fixing the pattern problem of the MDE neural network may increase the expressiveness of the evaluation.

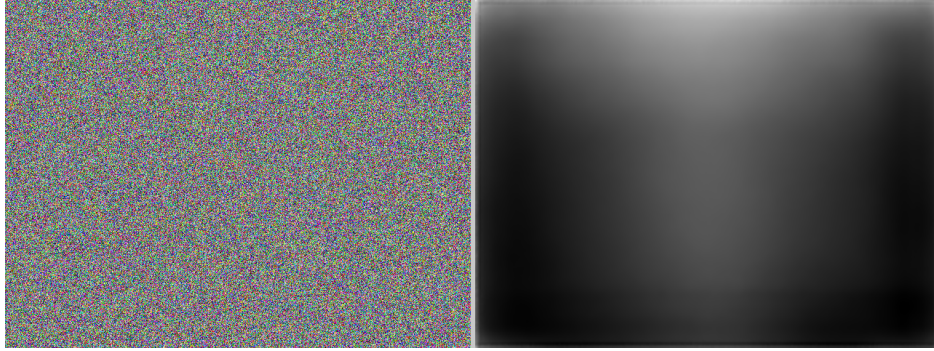


Figure 34: Feeding the neural network a randomly sampled RGB image (left) results in the depth estimation pattern on the right. The darker, the lower the depth. The brighter, the greater the depth.

4.7 Evaluation of SfM+MDE+Sky

This section evaluates the ensemble of the SfM, MDE, and Sky Segmentation approaches and its resulting combined occlusion map described in Section 3.2.4.

During this combined evaluation, previously derived hyperparameters are used: the SfM threshold is set to 30m, while MDE uses topographic Region Selection and MIN(UF,MAX) as Thresholding Method.

Looking at the improvement from SfM+Sky to SfM+MDE+Sky, the total mean FPR slightly increases by 0.1% from 6.66% to 6.76%, while the total mean FNR slightly decreases by 1.5% from 23.42% to 21.92% (see Table 3). However, MDE only adds depth information immediately around the wind turbine, the Cutout region. Since the Cutout makes up only a small percentage of the entire image, the total improvement is relatively small. But looking only at the Cutout (the only important region for occlusion since it contains the wind turbine), the improvement from SfM+Sky to SfM+MDE+Sky is significant since the mean FPR only increases by 1.05% from 5.95% to 7.00% while the mean FNR decreases by 35.00% from 62.48% to 27.48% (see Table 3).

Overall the ensemble correctly classifies 72.52% of all foreground and 93.00% of all background pixels inside the Cutout (see Table 3). Figure 35 shows the best occlusion

result for each sequence.

Algorithm	Mean FPR	Mean FNR
SfM 15m	0.4227	0.3806
SfM 30m	0.7648	0.1677
Sky Segmentation	0.0780	0.0757
SfM 30m + Sky	0.0666	0.2342
SfM 30m + MDE + Sky	0.0676	0.2192
SfM 30m + Sky (Cutout)	0.0595	0.6248
MDE + Sky (Cutout)	0.0527	0.3184
SfM 30m + MDE + Sky (Cutout)	0.0700	0.2748

Table 3: Mean FPR and FNR of the different approaches computed over all sequences.



Figure 35: Best occlusion result for each sequence. More results, including failure cases, are shown in the appendix (see Figures 39, 40, 41 and 42), and the entire dataset, including occlusion results and intermediate steps, can be accessed here: <https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset>.

5 Conclusion

Concluding, this thesis outlined the most common depth estimation techniques, evaluated them according to their use for the problem at hand, and proposed a solution by dividing the occlusion problem into Close-DDE and Long-DDE while applying different depth estimation techniques to both (see Section 2).

The proposed solution was implemented partly as a Unity Application using augmented reality Plug-ins like AR Foundation, ensuring compatibility for Android and iOS, and partly in Python using mainly OpenCV and TensorFlow. While the Unity Application was used to create a dataset containing 61 different scenes split into four sequences, the Python environment was used to perform the occlusion and analyze the results for which two metrics were introduced. Additionally, an efficient Sky-Segmentation-Algorithm has been proposed and implemented to improve the occlusion

results (see Section 3).

In the end, it could be shown that the best occlusion results were obtained when combining SfM for Close-DDE with MDE for Long-DDE while correcting misclassified sky regions with the proposed Sky-Segmentation-Algorithm. According to Section 4, the ensemble performs best with an SfM threshold of 30m and when MDE uses topographical information and the MIN(UF,MAX) Thresholding Method in order to generate the occlusion maps.

However, since the proposed Long-DDE solutions are computationally expensive, they cannot be used to recompute the occlusion map for every frame when deployed on a smartphone. As proposed in Section 2.3, a possibility would be to occasionally compute the Long-DDE occlusion map and adjust it according to the camera movement until it is recomputed. This way, the Long-DDE could either compute the occlusion map on a smartphone while splitting the computation over a few frames, or the occlusion map could be requested from a server that would take over the computation. This part, however, was not tested during this thesis due to the lack of time. Instead, the thesis focused on developing and evaluating a Sky-Segmentation-Algorithm because it became clear that SfM and MDE both suffered significantly from misclassified sky regions.

While this thesis did not result in a perfect solution, important insights have been generated, laying the foundation for future elaborations.

5.1 Limitations

Overall, the occlusion results have been rather mixed, and the numerical analysis has turned out to be complicated. Although the introduced metrics, FPR and FNR, are good estimators of how well an algorithm was able to classify its environment correctly, they do not reflect the quality of the occlusion result since the wind turbine makes up only a small percentage of the image. Therefore, if the algorithm performs extremely well on the entire image except for the region of the wind turbine, the FPR and FNR will be very good, while the visual occlusion result will be bad. Another reason for the lower expressiveness of the FPR and FNR is the dataset, which is rather small having 61 Scenes in its entirety, as well as the chosen ground truth labels, which could be more diverse (see Section 5.2). Therefore, when evaluating the approaches, one should keep in mind that sometimes visually better results have worse FPR and FNR and that deep knowledge and experience with the applied algorithms are needed for a proper evaluation.

While all investigated algorithms partially solve occlusion, they all have clear shortcomings. Both SfM and MDE suffer from sky regions and depend, therefore, on a Sky-Segmentation-Algorithm in order to yield reasonable occlusion results. In addition, SfM suffers from textureless surfaces and is unable to handle moving objects due

to the assumption of a static scene, which is needed to interpret the motion parallax effect. Furthermore, areas in which no depth could be estimated are inferred from depth estimations of neighboring pixels during the dense depth map computation, leading to wrong occlusions in areas that are probably too far away and should contain no depth at all.

On the other hand, the neural network, which was utilized for MDE, overfits to a specific pattern favoring smaller depths near the corners and is prone to wrongly estimating the depth of windows.

Furthermore, the DEM for Depth Estimation approach is too imprecise to be used for occlusion, although the camera position and rotation were adjusted manually, making it only suitable as topographic Region Selection for solving the relative value problem of the MDE neural network.

The Sky-Segmentation-Algorithm, which employs the simplified K-Means algorithm, relies on the assumption that a sky area containing mainly sky pixels can be selected based on the direction of the wind turbine (see Section 3.2.3). If the selected sky area does not consist primarily of sky pixels, the algorithm produces an incorrect sky segmentation that significantly impacts the overall occlusion result. Additionally, the Sky-Segmentation-Algorithm is prone to misclassifying distant, hazy outlined regions as sky areas, and its low resolution can result in imprecise occlusions in areas with intricate leaf and branch structures.

At last, another limitation occurs when using Unity in combination with AR Foundation for augmented reality on smartphones. While the smartphone front camera of an Honor 10 provides a resolution of 3456×4608 , the Camera Manager of AR Foundation only provides a camera image with a maximum resolution of 1440×1080 . This camera image is then rendered to the screen, during which the quality of the image is reduced once more (see Figure 36). Because of that, the visualization of wind farms becomes problematic for extremely far distances.



Figure 36: Left: Camera image acquired using the AR Foundation Camera Manager with the highest possible resolution. Middle: Image rendered on the screen inside the Unity AR Application. Right: Camera image taken by smartphone. The red-marked real-world wind turbine is around 7.24km away.

5.2 Future Work

One way to improve the analysis could be to enlarge the dataset with the provided Unity prototype. Alternatively, the ground truth maps could be enhanced by making the binary labeling more complex in form of a multi-class labeling, such as deciding between foreground pixels that SfM should detect and foreground pixels that are too far away for SfM, or sky and non-sky background pixels. Furthermore, in order to numerically represent the occlusion result, it could be analyzed which pixels of the wind turbine are correctly occluded instead of focusing on the entire image.

When it comes to the Sky-Segmentation-Algorithm, the next step would be to implement the proposed algorithm in Unity itself and try running it parallel to SfM on the smartphone. A problem of the Sky-Segmentation-Algorithm is that for each frame, the selected sky area used to compute the sky center must contain mainly sky pixels. This way, the user must always ensure that there is sky in this area. However, other techniques can be used to compute this sky center. For example, old sky centers could be used to smooth the sky center over time, or the user could manually identify sky regions during an initialization step.

Furthermore, a better way can be found to complement the SfM occlusion map by the sky segmentation. Instead of setting all detected sky pixels as background, the SfM and Sky confidences could be used to decide whether or not the sky segmentation is more confident than SfM. Additionally, a more robust variant of the proposed Sky-Segmentation-Algorithm can be deployed in combination with MDE to decrease misclassified sky regions.

Concerning the MDE neural network, transfer learning can be used to adjust the neural network to the case at hand and maybe fix the pattern problem described in Section 4.6.3. Another possibility would be to use GANs instead since they provide promising results, according to Reza et al. [32]. Another approach for MDE would be to use stereo deep learning networks, combining the monocular cues (except motion parallax) with binocular parallax. Furthermore, it could be seen during the evaluation that the edges/contours of the image are preserved inside the depth map. Those could be useful in combination with a segmentation algorithm instead of computing a simple threshold to yield an occlusion map.

Finally, this thesis computed the Long-DDE occlusion map for every single image. However, this approach is not feasible for a real-time augmented reality application. As discussed in Section 5, the practical use could potentially be achieved by occasional updates of the Long-DDE occlusion map.

References

- [1] M. Abrams, Y. Yamaguchi, and R. Crippen. ASTER GLOBAL DEM (GDEM) VERSION 3. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLIII-B4-2022:593–598, jun 2022. doi: 10.5194/isprs-archives-xliii-b4-2022-593-2022.
- [2] Ibraheem Alhashim and Peter Wonka. High quality monocular depth estimation via transfer learning. *arXiv*, abs/1812.11941, 2018. doi: 10.48550/ARXIV.1812.11941. Preprint.
- [3] Apple. ARKit, 2022. URL <https://developer.apple.com/augmented-reality/arkit/>. Accessed: 20-07-2022.
- [4] A. Bashfield and A. Keim. Continent-wide DEM creation for the European Union. In *International Symposium on Remote Sensing of Environment*, volume 34. ISPRS, apr 2011. URL <https://www.isprs.org/proceedings/2011/isrse-34/211104015Final00143.pdf>.
- [5] Andrew M. Colman. *A dictionary of psychology*. Oxford paperback reference. Oxford University Press, Oxford, 2. ed. edition, 2006. ISBN 9780192806321.
- [6] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Niessner. ScanNet: Richly-annotated 3D reconstructions of indoor scenes. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2432–2443. IEEE, jul 2017. doi: 10.1109/cvpr.2017.261.
- [7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, jun 2009. doi: 10.1109/cvpr.2009.5206848.
- [8] Hugh Durrant-Whyte, David Rye, and Eduardo Nebot. Localization of autonomous guided vehicles. In *Robotics Research*, pages 613–625. Springer London, 1996. URL https://doi.org/10.1007%2F978-1-4471-1021-7_69.
- [9] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in Neural Information Processing Systems (NIPS 2014)*, volume 27. Curran Associates, Inc., 2014. URL <https://proceedings.neurips.cc/paper/2014/file/7bccfde7714a1ebadf06c5f4cea752c1-Paper.pdf>.
- [10] EnBW Energie Baden-Württemberg AG. EnBW develops augmented reality app for visualisation of wind power plants, sep 2021. URL <https://www.enbw.com/company/press/augmented-reality-for-visualisation-of-wind-power-plant.html>. Accessed: 06-12-2022.

- [11] EXCIT3D GmbH. EXCIT3D develops augmented reality app for RWE renewables, oct 2021. URL <https://excit3d.de/en/excit3d-develops-augmented-reality-app-for-rwe-renewables/>. Accessed: 06-12-2022.
- [12] Tom G. Farr, Paul A. Rosen, Edward Caro, Robert Crippen, Riley Duren, Scott Hensley, Michael Kobrick, Mimi Paller, Ernesto Rodriguez, Ladislav Roth, David Seal, Scott Shaffer, Joanne Shimada, Jeffrey Umland, Marian Werner, Michael Oskin, Douglas Burbank, and Douglas Alsdorf. The shuttle radar topography mission. *Reviews of Geophysics*, 45(2), may 2007. doi: 10.1029/2005rg000183.
- [13] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002. ISBN 0130851981.
- [14] Huan Fu, Mingming Gong, Chaohui Wang, Kayhan Batmanghelich, and Dacheng Tao. Deep ordinal regression network for monocular depth estimation. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2002–2011. IEEE, jun 2018. doi: 10.1109/cvpr.2018.00214.
- [15] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on Information Theory*, 21(1):32–40, jan 1975. doi: 10.1109/tit.1975.1055330.
- [16] A Geiger, P Lenz, C Stiller, and R Urtasun. Vision meets robotics: The KITTI dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, aug 2013. doi: 10.1177/0278364913491297.
- [17] Google. ARCore, 2022. URL <https://developers.google.com/ar>. Accessed: 20-07-2022.
- [18] Google. Depth adds realism | ARCore | google developers, 2022. URL <https://developers.google.com/ar/develop/depth/>. Accessed: 20-07-2022.
- [19] Google. Build global-scale, immersive, location-based AR experiences with the ARCore geospatial api | google developers, 2022. URL <https://developers.google.com/ar/develop/geospatial>. Accessed: 06-12-2022.
- [20] S Grassi and T M Klein. 3D augmented reality for improving social acceptance and public participation in wind farms planning. *Journal of Physics: Conference Series*, 749:012020, sep 2016. doi: 10.1088/1742-6596/749/1/012020.
- [21] Laurence Hawker, Paul Bates, Jeffrey Neal, and Jonathan Rougier. Perspectives on digital elevation model (DEM) simulation for flood modeling in the absence of a high-accuracy open access global DEM. *Frontiers in Earth Science*, 6, dec 2018. doi: 10.3389/feart.2018.00233.

- [22] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. *arXiv*, 2016. doi: 10.48550/ARXIV.1608.06993. Preprint.
- [23] Michael Hölzl, Roland Neumeier, and Gerald Ostermayer. Analysis of compass sensor accuracy on several mobile devices in an industrial environment. In *Computer Aided Systems Theory - EUROCAST 2013*, pages 381–389. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-53862-9_49.
- [24] N. Kanopoulos, N. Vasanthavada, and R.L. Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of Solid-State Circuits*, 23(2):358–367, apr 1988. doi: 10.1109/4.996.
- [25] Marcel Kolvenbach and Nick Schader. Genehmigungen rückläufig: Flaute beim windkraftausbau, Jul 2022. URL <https://www.tagesschau.de/investigativ/swr/windkraftausbau-103.html>. Accessed: 06-12-2022.
- [26] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, mar 1982. doi: 10.1109/tit.1982.1056489.
- [27] Radu P. Mihail, Scott Workman, Zach Bessinger, and Nathan Jacobs. Sky segmentation in the wild: An empirical study. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–6. IEEE, mar 2016. doi: 10.1109/wacv.2016.7477637.
- [28] Quentin Mourcou, Anthony Fleury, Céline Franco, Frédéric Klopčič, and Nicolas Vuillerme. Performance evaluation of smartphone inertial sensors measurement for range of motion. *Sensors*, 15(9):23168–23187, sep 2015. doi: 10.3390/s150923168.
- [29] Kerry A. Nice, Jasper S. Wijnands, Ariane Middel, Jingcheng Wang, Yiming Qiu, Nan Zhao, Jason Thompson, Gideon D.P.A. Aschwanden, Haifeng Zhao, and Mark Stevenson. Sky pixel detection in outdoor imagery using an adaptive algorithm and machine learning. *Urban Climate*, 31:100572, mar 2020. doi: 10.1016/j.uclim.2019.100572.
- [30] Jinai Qu. Augmented reality for the visualization of wind farms. *Master Thesis at RWTH-Aachen, Theory of Hybrid Systems Informatik 2*, 2022.
- [31] Rambøll Group A/S. Virtual reality – a new digital solution for the visualisation of offshore wind projects, 2018. URL <https://ramboll.com/services-and-sectors/energy/wind-energy/virtual-reality-offshore-wind>. Accessed: 06-12-2022.
- [32] Md. Alimoor Reza, Jana Kosecka, and Philip David. FarSight: Long-range depth estimation from outdoor images. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4751–4757. IEEE, oct 2018. doi: 10.1109/iro.2018.8593971.

- [33] G. Riegler, S. D. Hennig, and M. Weber. WORLDDEM – a NOVEL GLOBAL FOUNDATION LAYER. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XL-3/W2:183–187, mar 2015. doi: 10.5194/isprsarchives-xl-3-w2-183-2015.
- [34] Ashutosh Saxena, Sung Chung, and Andrew Ng. Learning depth from single monocular images. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems (NIPS 2005)*, volume 18. MIT Press, 2005. URL <https://proceedings.neurips.cc/paper/2005/file/17d8da815fa21c57af9829fb0a869602-Paper.pdf>.
- [35] Ashutosh Saxena, Min Sun, and Andrew Y. Ng. Make3D: Learning 3D scene structure from a single still image. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):824–840, may 2009. doi: 10.1109/tpami.2008.132.
- [36] Bilawal Sidhu and Simon Lynen. VPS and the ARCore geospatial API, may 2022. URL <https://io.google/2022/program/fc42307b-3b0f-4925-bfea-99499ba08088/>. Accessed: 26-12-2022.
- [37] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from RGBD images. In *Computer Vision – ECCV 2012*, pages 746–760. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-33715-4_54.
- [38] Wesley E. Snyder and Hairong Qi. *Fundamentals of Computer Vision*. Cambridge University Press, 2017. doi: 10.1017/9781316882641.
- [39] Richard Szeliski. *Computer Vision, Algorithms and Applications, Second Edition*. Springer International Publishing, 2022. doi: 10.1007/978-3-030-34372-9.
- [40] Unity. Google ARCore XR Plug-in 5.0.3, 2022. URL <https://docs.unity3d.com/Packages/com.unity.xr.arcore@5.0/manual/index.html>. Accessed: 06-12-2022.
- [41] Unity. AR Foundation 5.0.3, 2022. URL <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.0/manual/index.html>. Accessed: 06-12-2022.
- [42] Unity. Apple ARKit XR Plug-in 5.0.3, 2022. URL <https://docs.unity3d.com/Packages/com.unity.xr.arkit@5.0/manual/index.html>. Accessed: 06-12-2022.
- [43] Unity. Unity manual, 2022. URL <https://docs.unity3d.com/Manual/>. Accessed: 22-12-2022.
- [44] Julien Valentin, Adarsh Kowdle, Jonathan T. Barron, Neal Wadhwa, Max Dzitsiuk, Michael Schoenberg, Vivek Verma, Ambrus Csaszar, Eric Turner, Ivan Dryanovski, Joao Afonso, Jose Pascoal, Konstantine Tsotsos, Mira Leung, Mirko

Schmidt, Onur Guleryuz, Sameh Khamis, Vladimir Tankovitch, Sean Fanello, Shahram Izadi, and Christoph Rhemann. Depth from motion for smartphone AR. *ACM Transactions on Graphics*, 37(6):1–19, dec 2018. doi: 10.1145/3272127.3275041.

- [45] Frank van Diggelen and Per Enge. The world’s first GPS MOOC and worldwide laboratory using smartphones. In *Proceedings of the 28th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2015)*, pages 361–369. ION Institute of Navigation, sep 2015.

6 Appendix

6.1 Graphs

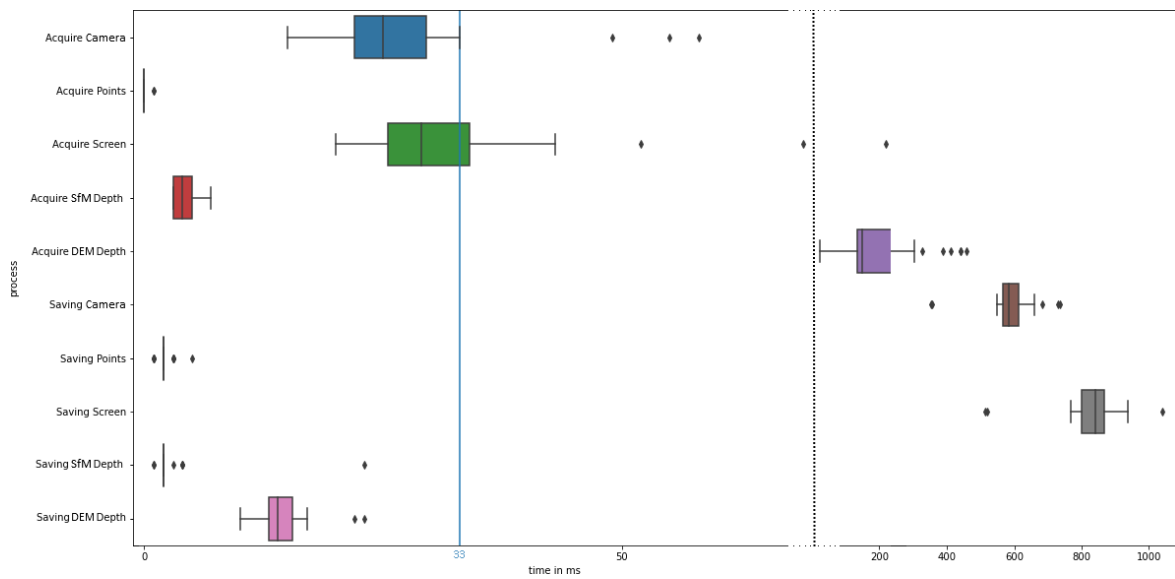


Figure 37: Visualization of times needed to complete certain processes. The blue line marks the maximally available time between frames to ensure a frame rate of 30fps. Data was captured using an Honor 10.

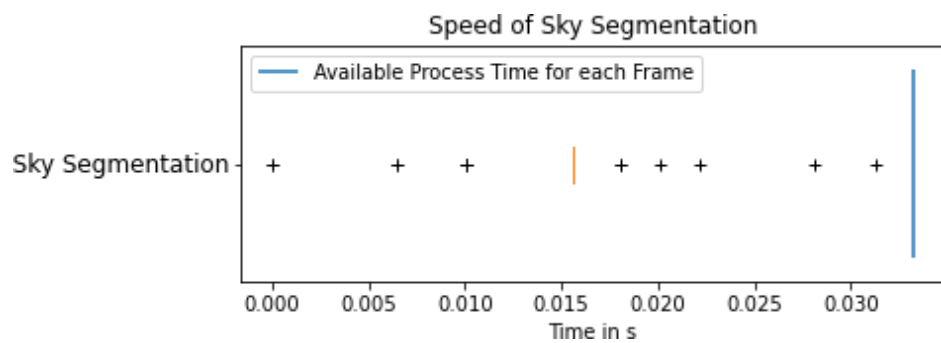


Figure 38: Boxplot of the observed timespans the Sky-Segmentation-Algorithm needs to segment a 120×160 image.

6.2 Paulusstraße Dataset

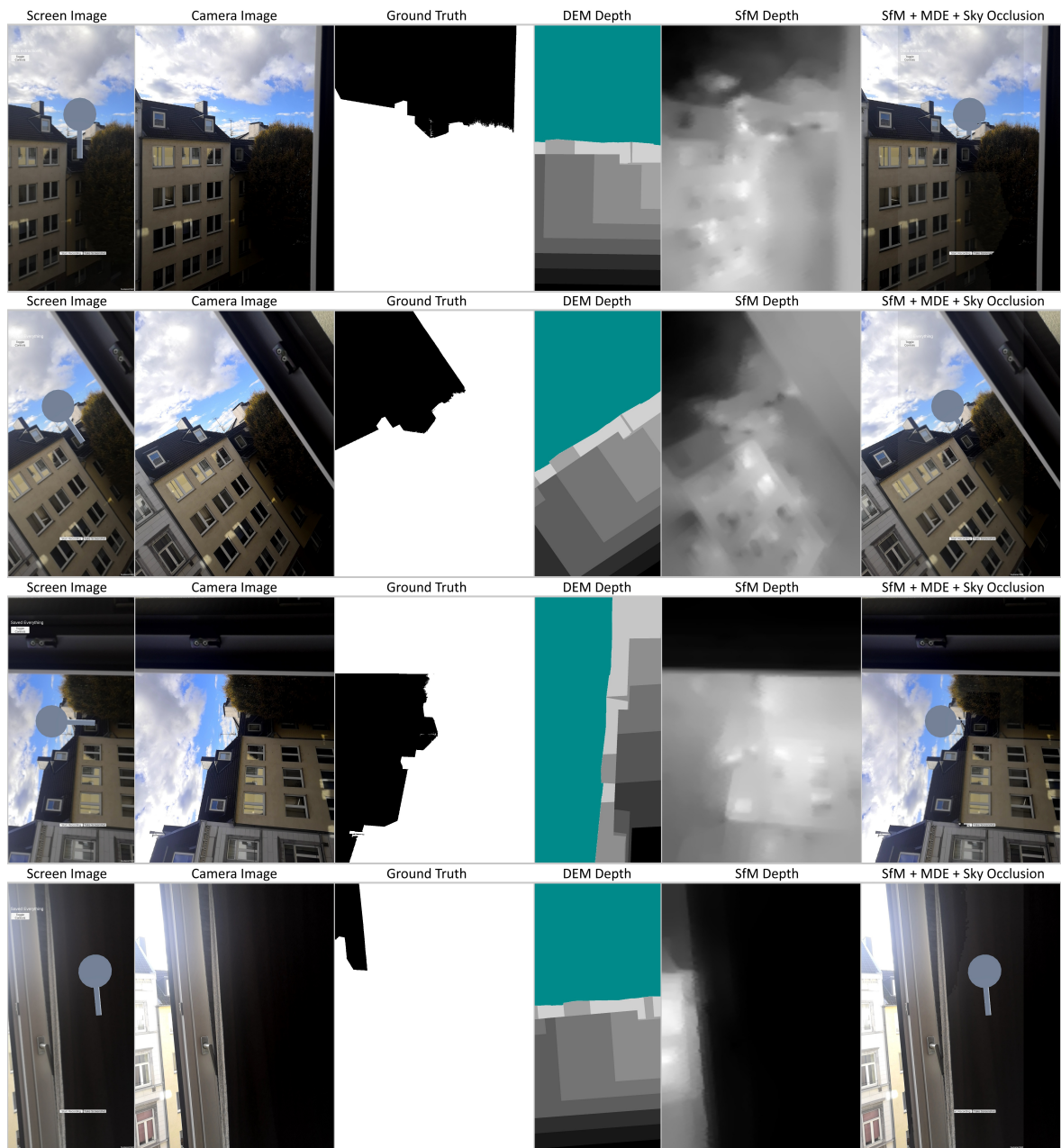


Figure 39: See <https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset> for entire dataset.

6.3 Königshügel Dataset

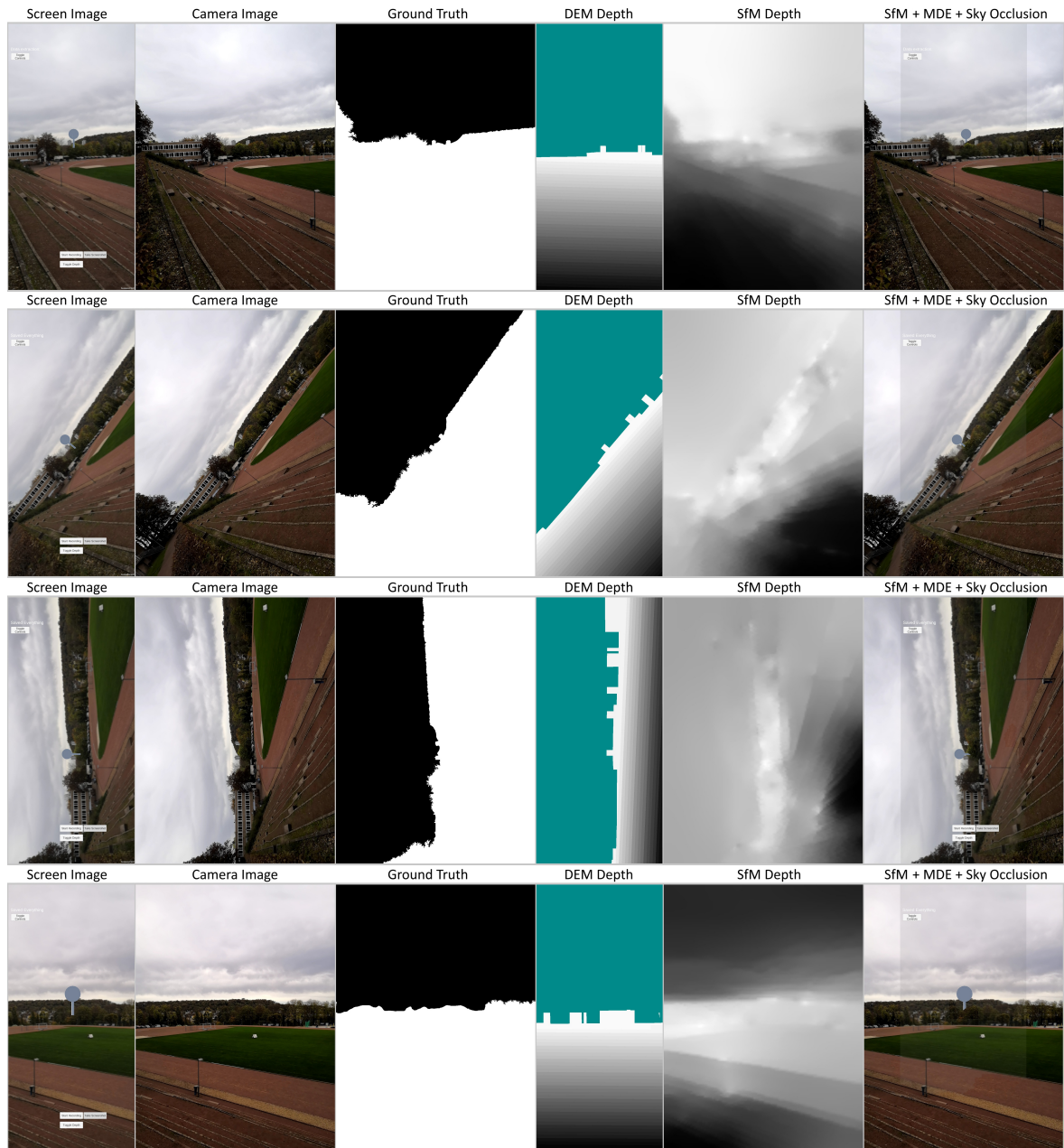


Figure 40: See <https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset> for entire dataset.

6.4 Informatikzentrum Dataset

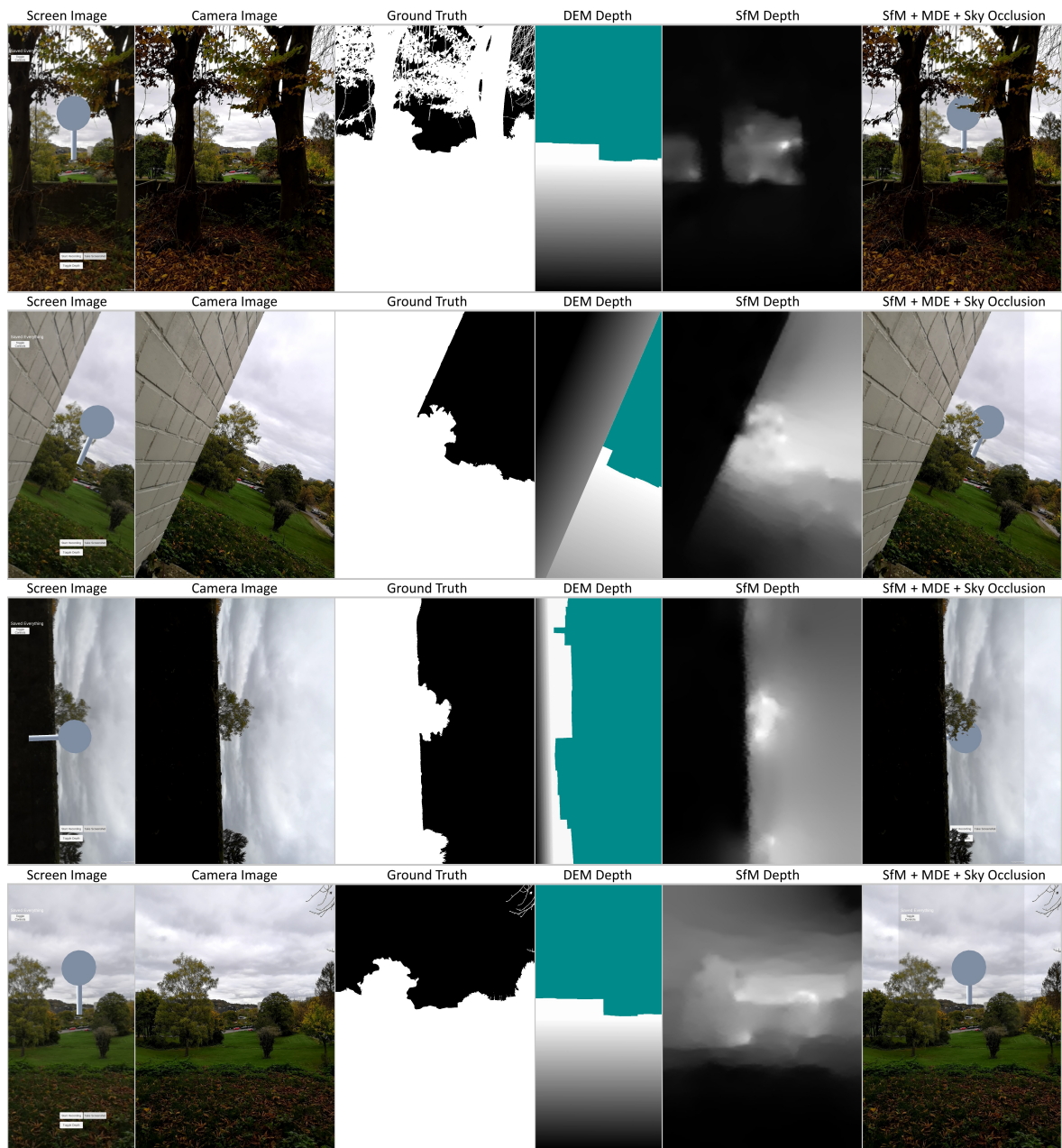


Figure 41: See <https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset> for entire dataset.

6.5 Lousberg Dataset

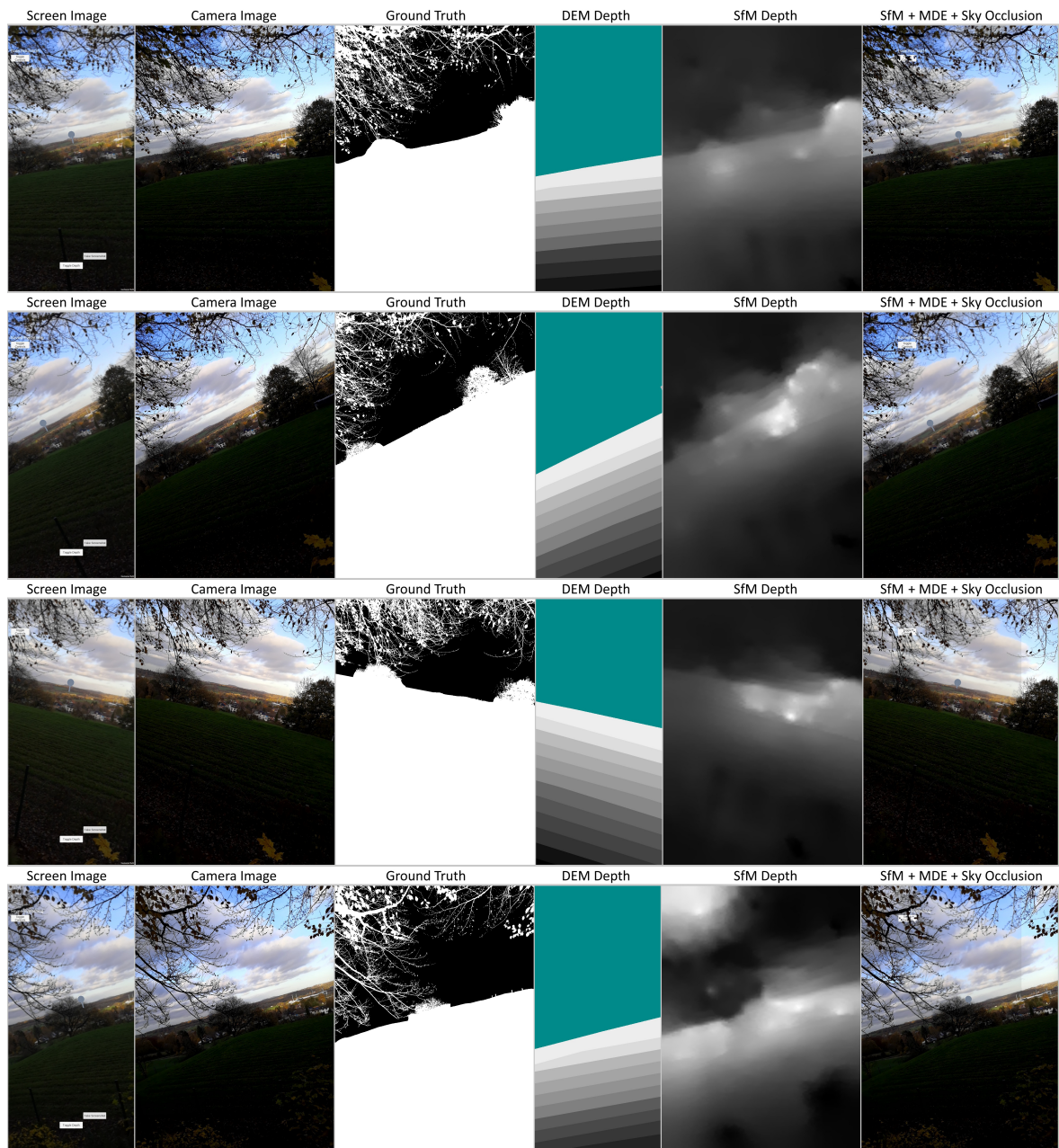


Figure 42: See <https://git.rwth-aachen.de/vgvalentino/windfarmocclusionpython/-/tree/master/Dataset> for entire dataset.

6.6 Shader

Faulty Occlusion Shader provided by ARCore XR Plug-in

```
1 Shader "Unlit/ARCoreBackgroundCopy/AfterOpagues"
2 {
3   Properties
4   {
5     _MainTex("Texture", 2D) = "white" {}
6     _EnvironmentDepth("Texture", 2D) = "black" {}
7   }
8
9   SubShader
10  {
11    Name "ARCore Background (After Opagues)"
12    Tags
13    {
14      "Queue" = "Background"
15      "RenderType" = "Background"
16      "ForceNoShadowCasting" = "True"
17    }
18
19    Pass
20    {
21      Name "ARCore Background Occlusion Handling"
22      Cull Off
23      ZTest GEqual
24      ZWrite On
25      Lighting Off
26      LOD 100
27      Tags
28      {
29        "LightMode" = "Always"
30      }
31
32      GLSLPROGRAM
33
34      #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
35
36      #pragma only_renderers gles3
37
38      #include "UnityCG.glslic"
39
40      #ifdef SHADER_API_GLES3
41      #extension GL_OES_EGL_image_external_essl3 : require
42      #endif // SHADER_API_GLES3
43
44      // Device display transform is provided by the AR Foundation camera background
45      // renderer.
46      uniform mat4 _UnityDisplayTransform;
47
48      #ifdef VERTEX
49      varying vec2 textureCoord;
50
51      void main()
52      {
53        #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
54          // Transform the position from object space to clip space.
55          gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
56
57          // Get quad uvs for sampling camera texture
58          textureCoordQuad = gl_MultiTexCoord0;
59
60          // Remap the texture coordinates based on the device rotation.
61          textureCoordEnvironment = (_UnityDisplayTransform * vec4(gl_MultiTexCoord0.x,
62            1.0f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
```

```

61 #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
62 }
63 #endif // VERTEX
64
65 #ifdef FRAGMENT
66     varying vec2 textureCoordQuad;
67     varying vec2 textureCoordEnvironment;
68     uniform float _UnityCameraForwardScale;
69
70 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
71     uniform sampler2D _EnvironmentDepth;
72     uniform sampler2D _CameraDepthTexture;
73 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
74
75     float ConvertDistanceToDepth(float d)
76     {
77         d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;
78
79         float zBufferParamsW = 1.0 / _ProjectionParams.y;
80         float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
81         float zBufferParamsX = 1.0 - zBufferParamsY;
82         float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
83
84         // Clip any distances smaller than the near clip plane, and compute the depth
85         // value from the distance.
86         return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) * ((1.0 / d)
87         - zBufferParamsW));
88     }
89
90     void main()
91     {
92         #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
93
94             vec3 result = texture(_MainTex, textureCoord).xyz;
95
96             float depth = texture(_CameraDepthTexture, textureCoordQuad).x;
97
98             float distance = texture(_EnvironmentDepth, textureCoordEnvironment).x;
99             float environmentDepth = ConvertDistanceToDepth(distance);
100
101             #ifndef UNITY_COLORSPACE_GAMMA
102                 result = GammaToLinearSpace(result);
103             #endif // !UNITY_COLORSPACE_GAMMA
104
105             if (depth >= environmentDepth)
106             {
107                 discard;
108             }
109
110             gl_FragColor = vec4(result, 1.0);
111             gl_FragDepth = depth;
112         #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
113     }
114 #endif // FRAGMENT
115
116     ENDGLSL
117 }
118
119 Pass
120 {
121     Name "AR Camera Background (ARCore)"
122     Cull Off
123     ZTest LEqual
124     ZWrite On
125     Lighting Off
126     LOD 100

```

```

125     Tags
126     {
127         "LightMode" = "Always"
128     }
129
130     GLSLPROGRAM
131
132     #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
133
134     #pragma only_renderers gles3
135
136     #include "UnityCG.glslinc"
137
138 #ifdef SHADER_API_GLES3
139 #extension GL_OES_EGL_image_external_essl3 : require
140 #endif // SHADER_API_GLES3
141
142     // Device display transform is provided by the AR Foundation camera background
143     // renderer.
144     uniform mat4 _UnityDisplayTransform;
145
146 #ifdef VERTEX
147     varying vec2 textureCoord;
148
149     void main()
150     {
151 #ifdef SHADER_API_GLES3
152         // Transform the position from object space to clip space.
153         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
154
155         // Remap the texture coordinates based on the device rotation.
156         textureCoord = (_UnityDisplayTransform * vec4(gl_MultiTexCoord0.x, 1.0f -
157             gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
158 #endif // SHADER_API_GLES3
159     }
160 #endif // VERTEX
161
162 #ifdef FRAGMENT
163     varying vec2 textureCoord;
164     uniform samplerExternalOES _MainTex;
165     uniform float _UnityCameraForwardScale;
166
167 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
168     uniform sampler2D _EnvironmentDepth;
169 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
170
171 #if defined(SHADER_API_GLES3) && !defined(UNITY_COLORSPACE_GAMMA)
172     float GammaToLinearSpaceExact (float value)
173     {
174         if (value <= 0.04045F)
175             return value / 12.92F;
176         else if (value < 1.0F)
177             return pow((value + 0.055F)/1.055F, 2.4F);
178         else
179             return pow(value, 2.2F);
180     }
181
182     vec3 GammaToLinearSpace (vec3 sRGB)
183     {
184         // Approximate version from http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
185         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) + 0.012522878F);
186
187         // Precise version, useful for debugging, but the pow() function is too slow.
188         // return vec3(GammaToLinearSpaceExact(sRGB.r), GammaToLinearSpaceExact(sRGB.g), GammaToLinearSpaceExact(sRGB.b));
189     }
190 #endif
191

```



```

187     }
188
189 #endif // SHADER_API_GLES3 && !UNITY_COLORSPACE_GAMMA
190
191     float ConvertDistanceToDepth(float d)
192     {
193         d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;
194
195         float zBufferParamsW = 1.0 / _ProjectionParams.y;
196         float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
197         float zBufferParamsX = 1.0 - zBufferParamsY;
198         float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
199
200         // Clip any distances smaller than the near clip plane, and compute the depth
201         // value from the distance.
202         return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) * ((1.0 / d)
203         - zBufferParamsW));
204     }
205
206     void main()
207     {
208 #ifdef SHADER_API_GLES3
209         vec3 result = texture(_MainTex, textureCoord).xyz;
210         float depth = 1.0;
211
212 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
213         float distance = texture(_EnvironmentDepth, textureCoord).x;
214         depth = ConvertDistanceToDepth(distance);
215 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
216
217 #ifndef UNITY_COLORSPACE_GAMMA
218         result = GammaToLinearSpace(result);
219 #endif // !UNITY_COLORSPACE_GAMMA
220
221         gl_FragColor = vec4(result, 1.0);
222         gl_FragDepth = depth;
223 #endif // SHADER_API_GLES3
224     }
225 #endif // FRAGMENT
226 ENDGLSL
227 }
228
229 FallBack Off
230 }

```

Shader 1: ARCoreBackgroundAfterOpques.shader

Fix of Faulty Occlusion Shader provided by ARCore XR Plug-in

```

1 Shader "Unlit/ARCoreOcclusionFix"
2 {
3     Properties
4     {
5         _MainTex("Texture", 2D) = "white" {}
6         _EnvironmentDepth("Texture", 2D) = "black" {}
7     }
8
9     SubShader
10    {
11        Name "ARCore Background (After Opques)"
12        Tags
13        {

```

```

14         "Queue" = "Background"
15         "RenderType" = "Background"
16         "ForceNoShadowCasting" = "True"
17     }
18
19     Pass
20     {
21         Name "ARCore Background Occlusion Handling"
22         Cull Off
23         ZTest GEqual
24         ZWrite On
25         Lighting Off
26         LOD 100
27         Tags
28         {
29             "LightMode" = "Always"
30         }
31
32         GLSLPROGRAM
33
34         #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
35         #pragma only_renderers gles3
36         #include "UnityCG.glslinc"
37
38     #ifdef SHADER_API_GLES3
39     #extension GL_OES_EGL_image_external_essl3 : require
40     #endif // SHADER_API_GLES3
41
42         // Device display transform is provided by the AR Foundation camera
43         background renderer.
44         uniform mat4 _UnityDisplayTransform;
45
46     #ifdef VERTEX
47         varying vec2 textureCoordQuad;
48         varying vec2 textureCoordEnvironment;
49
50         void main()
51         {
52     #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
53             // Transform the position from object space to clip space.
54             gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
55
56             // Get quad uvs for sampling camera texture
57             textureCoordQuad = vec2(gl_MultiTexCoord0.x, gl_MultiTexCoord0.y);
58
59             // Remap the texture coordinates based on the device rotation.
60             textureCoordEnvironment = (_UnityDisplayTransform * vec4(
61                 gl_MultiTexCoord0.x, 1.0f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
62     #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
63         }
64     #endif // VERTEX
65
66     #ifdef FRAGMENT
67         varying vec2 textureCoordQuad;
68         varying vec2 textureCoordEnvironment;
69         uniform float _UnityCameraForwardScale;
70         uniform samplerExternalOES _MainTex;
71
72     #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
73         uniform sampler2D _EnvironmentDepth;
74         uniform sampler2D _CameraDepthTexture;
75     #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
76
77         float ConvertDistanceToDepth(float d)
78         {
79             d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;

```

```

78
79         float zBufferParamsW = 1.0 / _ProjectionParams.y;
80         float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
81         float zBufferParamsX = 1.0 - zBufferParamsY;
82         float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
83
84         // Clip any distances smaller than the near clip plane, and compute
the depth value from the distance.
85         return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) *
((1.0 / d) - zBufferParamsW));
86     }
87     vec3 GammaToLinearSpace(vec3 sRGB)
88     {
89         // Approximate version from http://chilliant.blogspot.com.au/2012/08/
srgb-approximations-for-hlsl.html?m=1
90         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
0.012522878F);
91
92         // Precise version, useful for debugging, but the pow() function is
too slow.
93         // return vec3(GammaToLinearSpaceExact(sRGB.r),
GammaToLinearSpaceExact(sRGB.g), GammaToLinearSpaceExact(sRGB.b));
94     }
95
96     void main()
97     {
98 #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
99
100         vec3 result = texture(_MainTex, textureCoordEnvironment).xyz;
101
102         float depth = texture(_CameraDepthTexture, textureCoordQuad).x;
103
104         float distance = texture(_EnvironmentDepth, textureCoordEnvironment).x
;
105         float environmentDepth = ConvertDistanceToDepth(distance);
106
107 #ifndef UNITY_COLORSPACE_GAMMA
108         result = GammaToLinearSpace(result);
109 #endif // !UNITY_COLORSPACE_GAMMA
110         if (depth >= environmentDepth)
111         {
112             discard;
113         }
114         gl_FragColor = vec4(result, 1.0);
115         gl_FragDepth = depth;
116 #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
117     }
118
119 #endif // FRAGMENT
120     ENDGLSL
121 }
122
123 Pass
124 {
125     Name "AR Camera Background (ARCore)"
126     Cull Off
127     ZTest LEqual
128     ZWrite On
129     Lighting Off
130     LOD 100
131     Tags
132     {
133         "LightMode" = "Always"
134     }
135
136     GLSLPROGRAM

```

```

137
138     #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
139
140     #pragma only_renderers gles3
141
142     #include "UnityCG.glsinc"
143
144 #ifndef SHADER_API_GLES3
145 #extension GL_OES_EGL_image_external_essl3 : require
146 #endif // SHADER_API_GLES3
147
148 // Device display transform is provided by the AR Foundation camera background
149 // renderer.
150 uniform mat4 _UnityDisplayTransform;
151
152 #ifdef VERTEX
153     varying vec2 textureCoord;
154
155     void main()
156     {
157 #ifdef SHADER_API_GLES3
158         // Transform the position from object space to clip space.
159         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
160
161         // Remap the texture coordinates based on the device rotation.
162         textureCoord = (_UnityDisplayTransform * vec4(gl_MultiTexCoord0.x, 1.0
163             f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
164 #endif // SHADER_API_GLES3
165     }
166 #endif // VERTEX
167
168 #ifdef FRAGMENT
169     varying vec2 textureCoord;
170     uniform samplerExternalOES _MainTex;
171     uniform float _UnityCameraForwardScale;
172
173 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
174     uniform sampler2D _EnvironmentDepth;
175 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
176
177 #if defined(SHADER_API_GLES3) && !defined(UNITY_COLORSPACE_GAMMA)
178     float GammaToLinearSpaceExact(float value)
179     {
180         if (value <= 0.04045F)
181             return value / 12.92F;
182         else if (value < 1.0F)
183             return pow((value + 0.055F) / 1.055F, 2.4F);
184         else
185             return pow(value, 2.2F);
186     }
187
188     vec3 GammaToLinearSpace(vec3 sRGB)
189     {
190         // Approximate version from http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
191         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
192             0.012522878F);
193
194         // Precise version, useful for debugging, but the pow() function is
195         // too slow.
196         // return vec3(GammaToLinearSpaceExact(sRGB.r),
197             GammaToLinearSpaceExact(sRGB.g), GammaToLinearSpaceExact(sRGB.b));
198     }
199 #endif // SHADER_API_GLES3 && !UNITY_COLORSPACE_GAMMA
200
201 #endif

```

```

197         float ConvertDistanceToDepth(float d)
198         {
199             d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;
200
201             float zBufferParamsW = 1.0 / _ProjectionParams.y;
202             float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
203             float zBufferParamsX = 1.0 - zBufferParamsY;
204             float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
205
206             // Clip any distances smaller than the near clip plane, and compute
the depth value from the distance.
207             return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) *
((1.0 / d) - zBufferParamsW));
208         }
209
210         void main()
211         {
212 #ifdef SHADER_API_GLES3
213             vec3 result = texture(_MainTex, textureCoord).xyz;
214             float depth = 1.0;
215
216 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
217             float distance = texture(_EnvironmentDepth, textureCoord).x;
218             depth = ConvertDistanceToDepth(distance);
219 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
220
221 #ifndef UNITY_COLORSPACE_GAMMA
222             result = GammaToLinearSpace(result);
223 #endif // !UNITY_COLORSPACE_GAMMA
224
225             gl_FragColor = vec4(result, 1.0);
226             gl_FragDepth = depth;
227 #endif // SHADER_API_GLES3
228         }
229
230 #endif // FRAGMENT
231     ENDGLSL
232 }
233 }
234
235     FallBack Off
236 }

```

Shader 2: ARCoreOcclusionFix.shader

Threshold Shader

```

1 Shader "Unlit/ARCoreOcclusionThreshold"
2 {
3     Properties
4     {
5         _MainTex("Texture", 2D) = "white" {}
6         _EnvironmentDepth("Texture", 2D) = "black" {}
7     }
8     SubShader
9     {
10         Name "ARCore Background (After Opaques)"
11         Tags
12         {
13             "Queue" = "Background"
14             "RenderType" = "Background"
15             "ForceNoShadowCasting" = "True"
16         }
17         Pass

```

```

18     {
19         Name "ARCore Background Occlusion Handling"
20         Cull Off
21         ZTest GEqual
22         ZWrite On
23         Lighting Off
24         LOD 100
25         Tags
26         {
27             "LightMode" = "Always"
28         }
29
30         GLSLPROGRAM
31         #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
32         #pragma only_renderers gles3
33         #include "UnityCG.glslinc"
34
35     #ifdef SHADER_API_GLES3
36     #extension GL_OES_EGL_image_external_essl3 : require
37     #endif // SHADER_API_GLES3
38
39     // Device display transform is provided by the AR Foundation camera background
40     // renderer.
41     uniform mat4 _UnityDisplayTransform;
42
43     #ifdef VERTEX
44         varying vec2 textureCoordQuad;
45         varying vec2 textureCoordEnvironment;
46
47         void main()
48         {
49             #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
50                 // Transform the position from object space to clip space.
51                 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
52
53                 // Get quad uvs for sampling camera texture
54                 textureCoordQuad = vec2(gl_MultiTexCoord0.x, gl_MultiTexCoord0.y);
55
56                 // Remap the texture coordinates based on the device rotation.
57                 textureCoordEnvironment = (_UnityDisplayTransform * vec4(
58                     gl_MultiTexCoord0.x, 1.0f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
59             #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
60         }
61     #endif // VERTEX
62
63     #ifdef FRAGMENT
64         varying vec2 textureCoordQuad; // Unity Camera Depth Texture
65         Coordinates
66         varying vec2 textureCoordEnvironment; // Environment Depth and
67         Camera Texture Coordinates
68         uniform float _UnityCameraForwardScale;
69         uniform samplerExternalOES _MainTex; // Camera Texture
70
71     #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
72         uniform sampler2D _EnvironmentDepth; // Environment Depth Texture
73         uniform sampler2D _CameraDepthTexture; // Unity Camera Depth Texture
74     #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
75
76         float ConvertDistanceToDepth(float d)
77         {
78             d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;
79
80             float zBufferParamsW = 1.0 / _ProjectionParams.y;
81             float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
82             float zBufferParamsX = 1.0 - zBufferParamsY;
83             float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;

```



```

80
81         // Clip any distances smaller than the near clip plane, and compute
the depth value from the distance.
82         return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) *
((1.0 / d) - zBufferParamsW));
83     }
84     vec3 GammaToLinearSpace(vec3 sRGB)
85     {
86         // Approximate version from http://chilliant.blogspot.com.au/2012/08/
srgb-approximations-for-hlsl.html?m=1
87         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
0.012522878F);
88     }
89
90     void main()
91     {
92 #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
93
94         vec3 result = texture(_MainTex, textureCoordEnvironment).xyz;
95         float depth = texture(_CameraDepthTexture, textureCoordQuad).x;
96         float distance = texture(_EnvironmentDepth, textureCoordEnvironment).x
;
97         float environmentDepth = ConvertDistanceToDepth(distance);
98
99 #ifndef UNITY_COLORSPACE_GAMMA
100         result = GammaToLinearSpace(result);
101 #endif // !UNITY_COLORSPACE_GAMMA
102         if (depth >= environmentDepth)
103         {
104             discard;
105         }
106         gl_FragColor = vec4(result, 1.0);
107         gl_FragDepth = depth;
108 #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
109     }
110 #endif // FRAGMENT
111     ENDGLSL
112 }
113
114 Pass
115 {
116     Name "AR Camera Background (ARCore)"
117     Cull Off
118     ZTest LEqual
119     ZWrite On
120     Lighting Off
121     LOD 100
122     Tags
123     {
124         "LightMode" = "Always"
125     }
126
127     GLSLPROGRAM
128     #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
129     #pragma only_renderers gles3
130     #include "UnityCG.glslinc"
131
132 #ifdef SHADER_API_GLES3
133 #extension GL_OES_EGL_image_external_essl3 : require
134 #endif // SHADER_API_GLES3
135
136 // Device display transform is provided by the AR Foundation camera background
renderer.
137 uniform mat4 _UnityDisplayTransform;
138
139 #ifdef VERTEX

```

```

140         varying vec2 textureCoord;
141
142         void main()
143         {
144 #ifdef SHADER_API_GLES3
145             // Transform the position from object space to clip space.
146             gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
147
148             // Remap the texture coordinates based on the device rotation.
149             textureCoord = (_UnityDisplayTransform * vec4(gl_MultiTexCoord0.x, 1.0
150 f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
151 #endif // SHADER_API_GLES3
152         }
153 #endif // VERTEX
154
155 #ifdef FRAGMENT
156     varying vec2 textureCoord; // Environment Depth and
157     Camera Texture Coordinates
158     uniform samplerExternalOES _MainTex; // Camera Texture
159     uniform float _UnityCameraForwardScale;
160
161 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
162     uniform sampler2D _EnvironmentDepth; // Environment Depth Texture
163 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
164
165 #if defined(SHADER_API_GLES3) && !defined(UNITY_COLORSPACE_GAMMA)
166     vec3 GammaToLinearSpace(vec3 sRGB)
167     {
168         // Approximate version from http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
169         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
170 0.012522878F);
171     }
172 #endif // SHADER_API_GLES3 && !UNITY_COLORSPACE_GAMMA
173
174 float ConvertDistanceToDepth(float d)
175 {
176     d = _UnityCameraForwardScale > 0.0 ? _UnityCameraForwardScale * d : d;
177
178     float zBufferParamsW = 1.0 / _ProjectionParams.y;
179     float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
180     float zBufferParamsX = 1.0 - zBufferParamsY;
181     float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
182
183     // Clip any distances smaller than the near clip plane, and compute
184     the depth value from the distance.
185     return (d < _ProjectionParams.y) ? 1.0f : ((1.0 / zBufferParamsZ) *
186 ((1.0 / d) - zBufferParamsW));
187 }
188
189 void main()
190 {
191 #ifdef SHADER_API_GLES3
192     vec3 result = texture(_MainTex, textureCoord).xyz;
193     float threshold = 15.0; // threshold set to 15m after that ignore
194     depth and set it always to 1.0
195     float depth = 1.0; // Make sure that the new geometry is always
196     overdrawn (if depth=0.0 the new geometry overdraws everything else)
197
198 #ifdef ARCORE_ENVIRONMENT_DEPTH_ENABLED
199     float distance = texture(_EnvironmentDepth, textureCoord).x;
200
201     if (distance <= threshold) {
202         depth = ConvertDistanceToDepth(distance);
203     }
204 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED

```

```

198
199 #ifndef UNITY_COLORSPACE_GAMMA
200         result = GammaToLinearSpace(result);
201 #endif // !UNITY_COLORSPACE_GAMMA
202
203         gl_FragColor = vec4(result, 1.0);
204         gl_FragDepth = depth;
205 #endif // SHADER_API_GLES3
206     }
207
208 #endif // FRAGMENT
209     ENDGLSL
210 }
211 }
212     FallBack Off
213 }

```

Shader 3: ACCoreOcclusionThreshold.shader

Shader used for DEMs for Depth Estimation

```

1 Shader "Unlit/RenderGeometryDepth"
2 {
3     Properties {}
4
5     SubShader
6     {
7         Name "WindfarmOcclusion Topographic Depth Visualization"
8         Tags
9         {
10             "Queue" = "Geometry" // Tells Unity when this shader shall
11             be applied.
12             "RenderType" = "Geometry"
13             "ForceNoShadowCasting" = "True" // Tells Unity to prevent geometry in
14             this Subshader to cast shadows.
15         }
16
17         Pass
18         {
19             Name "WindfarmOcclusion Topographic Depth Visualization"
20             Cull Off
21             ZTest LEqual
22             ZWrite On
23             Lighting Off
24             LOD 100
25             Tags
26             {
27                 "LightMode" = "Always"
28             }
29
30             GLSLPROGRAM
31
32             #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
33             #pragma only_renderers gles3
34             #include "UnityCG.glslinc"
35
36 #ifdef SHADER_API_GLES3
37 #extension GL_OES_EGL_image_external_essl3 : require
38 #endif // SHADER_API_GLES3
39
40 #ifdef VERTEX
41     varying vec2 textureCoordQuad;
42     void main()

```

```

41     {
42 #ifdef SHADER_API_GLES3
43     // Transform the position from object space to clip space.
44     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
45     textureCoordQuad = vec2(gl_MultiTexCoord0.x, gl_MultiTexCoord0.y);
46 #endif // SHADER_API_GLES3
47     }
48 #endif // VERTEX
49
50 #ifdef FRAGMENT
51     varying vec2 textureCoordQuad;
52     uniform sampler2D _CameraDepthTexture;
53 #if defined(SHADER_API_GLES3) && !defined(UNITY_COLORSPACE_GAMMA)
54     vec3 GammaToLinearSpace (vec3 sRGB)
55     {
56         // Approximate version from http://chilliant.blogspot.com.au/2012/08/
57         // srgb-approximations-for-hlsl.html?m=1
58         return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
59             0.012522878F);
60     }
61 #endif // SHADER_API_GLES3 && !UNITY_COLORSPACE_GAMMA
62
63     float LinearEyeDepth(float depth)
64     {
65         float zBufferParamsW = 1.0 / _ProjectionParams.y;
66         float zBufferParamsY = _ProjectionParams.z * zBufferParamsW;
67         float zBufferParamsX = 1.0 - zBufferParamsY;
68         float zBufferParamsZ = zBufferParamsX * _ProjectionParams.w;
69         return 1.0 / (zBufferParamsZ * depth + zBufferParamsW);
70     }
71
72     void main()
73     {
74 #ifdef SHADER_API_GLES3
75         float depth = texture(_CameraDepthTexture, textureCoordQuad).x;
76         float distance = LinearEyeDepth(depth);
77         vec3 result = vec3(1.0 / distance, 0.0, 0.0);
78 #ifndef UNITY_COLORSPACE_GAMMA
79         result = GammaToLinearSpace(result);
80 #endif // !UNITY_COLORSPACE_GAMMA
81         gl_FragColor = vec4(result, 1.0);
82         gl_FragDepth = 0.0; // Make sure that the new geometry overdraws
83         // whats already drawn to screen.
84 #endif // SHADER_API_GLES3
85     }
86 #endif // FRAGMENT
87     ENDGLSL
88 }

```

Shader 4: RenderGeometryDepth.shader

Shader used to not occlude objects for proper Data Extraction

```

1 Shader "Unlit/ARCoreNoOcclusionForAnalysis"
2 {
3     Properties
4     {
5         _MainTex("Texture", 2D) = "white" {}
6     }
7
8     SubShader

```

```

9      {
10          Name "ARCore Background (After Opaques)"
11          Tags
12          {
13              "Queue" = "Background"
14              "RenderType" = "Background"
15              "ForceNoShadowCasting" = "True"
16          }
17          Pass
18          {
19              Name "AR Camera Background (ARCore)"
20              Cull Off
21              ZTest LEqual
22              ZWrite On
23              Lighting Off
24              LOD 100
25              Tags
26              {
27                  "LightMode" = "Always"
28              }
29
30              GLSLPROGRAM
31
32              #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
33              #pragma only_renderers gles3
34              #include "UnityCG.glslinc"
35
36              #ifndef SHADER_API_GLES3
37              #extension GL_OES_EGL_image_external_essl3 : require
38              #endif // SHADER_API_GLES3
39
40              // Device display transform is provided by the AR Foundation camera background
41              // renderer.
42              uniform mat4 _UnityDisplayTransform;
43
44              #ifndef VERTEX
45              varying vec2 textureCoord;
46
47              void main()
48              {
49                  #ifndef SHADER_API_GLES3
50                      // Transform the position from object space to clip space.
51                      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
52
53                      // Remap the texture coordinates based on the device rotation.
54                      textureCoord = (_UnityDisplayTransform * vec4(gl_MultiTexCoord0.x, 1.0
55                          f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
56                  #endif // SHADER_API_GLES3
57              }
58              #endif // VERTEX
59
60              #ifndef FRAGMENT
61              varying vec2 textureCoord;
62              uniform samplerExternalOES _MainTex;
63              uniform float _UnityCameraForwardScale;
64
65              #if defined(SHADER_API_GLES3) && !defined(UNITY_COLORSPACE_GAMMA)
66                  vec3 GammaToLinearSpace(vec3 sRGB)
67                  {
68                      // Approximate version from http://chilliant.blogspot.com.au/2012/08/srgb-approximations-for-hlsl.html?m=1
69                      return sRGB * (sRGB * (sRGB * 0.305306011F + 0.682171111F) +
70                          0.012522878F);
71                  }
72              #endif // SHADER_API_GLES3 && !UNITY_COLORSPACE_GAMMA

```

```

71         void main()
72         {
73 #ifdef SHADER_API_GLES3
74             vec3 result = texture(_MainTex, textureCoord).xyz;
75 #ifndef UNITY_COLORSPACE_GAMMA
76             result = GammaToLinearSpace(result);
77 #endif // !UNITY_COLORSPACE_GAMMA
78             gl_FragColor = vec4(result, 1.0);
79             gl_FragDepth = 1.0; // Always behind existing geometry.
80 #endif // SHADER_API_GLES3
81         }
82 #endif // FRAGMENT
83     ENDGLSL
84 }
85 }
86 FallBack Off
87 }

```

Shader 5: ARCoreNoOcclusionForAnalysis.shader

Shader used to render ARCore Depth to screen

```

1 Shader "Unlit/ARCoreVisualizeEnvironmentDepth"
2 {
3     Properties
4     {
5         _MainTex("Texture", 2D) = "white" {}
6         _EnvironmentDepth("Texture", 2D) = "black" {}
7     }
8     SubShader
9     {
10        Name "ARCore Background (After Opaques)"
11        Tags
12        {
13            "Queue" = "Background"
14            "RenderType" = "Background"
15            "ForceNoShadowCasting" = "True"
16        }
17        Pass
18        {
19            Name "ARCore Background Occlusion Handling"
20            Cull Off
21            ZTest LEqual
22            ZWrite On
23            Lighting Off
24            LOD 100
25            Tags
26            {
27                "LightMode" = "Always"
28            }
29        }
30        GLSLPROGRAM
31
32        #pragma multi_compile_local __ ARCORE_ENVIRONMENT_DEPTH_ENABLED
33        #pragma only_renderers gles3
34        #include "UnityCG.glslic"
35
36 #ifdef SHADER_API_GLES3
37 #extension GL_OES_EGL_image_external_essl3 : require
38 #endif // SHADER_API_GLES3
39
40 // Device display transform is provided by the AR Foundation camera background
41 // renderer.
42 uniform mat4 _UnityDisplayTransform;

```

```

42
43 #ifdef VERTEX
44     varying vec2 textureCoordEnvironment;
45
46     void main()
47     {
48 #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
49         // Transform the position from object space to clip space.
50         gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
51
52         // Remap the texture coordinates based on the device rotation.
53         textureCoordEnvironment = (_UnityDisplayTransform * vec4(
54             gl_MultiTexCoord0.x, 1.0f - gl_MultiTexCoord0.y, 1.0f, 0.0f)).xy;
55 #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
56     }
57 #endif // VERTEX
58
59 #ifdef FRAGMENT
60     varying vec2 textureCoordEnvironment;
61     uniform float _UnityCameraForwardScale;
62
63 #if defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
64     uniform sampler2D _EnvironmentDepth;
65 #endif // ARCORE_ENVIRONMENT_DEPTH_ENABLED
66
67     void main()
68     {
69 #if defined(SHADER_API_GLES3) && defined(ARCORE_ENVIRONMENT_DEPTH_ENABLED)
70
71         float distance = texture(_EnvironmentDepth, textureCoordEnvironment).x
72         ;
73
74 #ifdef UNITY_COLORSPACE_GAMMA
75         result = GammaToLinearSpace(result);
76 #endif // !UNITY_COLORSPACE_GAMMA
77         //gl_FragColor = vec4(1.0 / distance, 0.0, 0.0, 1.0);
78         float red = 0.0;
79         float green = 0.0;
80         float blue = 0.0;
81         if (distance <= 20.0)
82         {
83             green = 1.0/distance;
84         }
85         else if (distance <= 40.0) {
86             blue = 1.0/(distance - 20.0);
87         }
88         else {
89             red = 1.0/(distance - 40.0);
90         }
91         //gl_FragColor = vec4(distance/65.0, 0.0, 0.0, 1.0);
92         gl_FragColor = vec4(red, green, blue, 1.0);
93         gl_FragDepth = 0.0;
94 #endif // SHADER_API_GLES3 && ARCORE_ENVIRONMENT_DEPTH_ENABLED
95     }
96 #endif // FRAGMENT
97     ENDGLSL
98 }
99     FallBack Off
100 }

```

Shader 6: ARCoreVisualizeEnvironmentDepth.shader