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

Augmented Reality für die Visualisierung von Windparks mit Hinderniserkennung

Augmented Reality for the Visualization of Wind Farms with Obstacle Detection

Bachelorarbeit
Informatik

August 2022

Vorgelegt von Presented by	Patrick Chrestin Matrikelnummer: 402502 patrick.chrestin@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. rer. nat. Pascal Richter Lehr- und Forschungsgebiet: Theorie der hybriden Systeme RWTH Aachen University

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Chrestin, Patrick — 402502

Name, Vorname/Last Name, First Name — Matrikelnummer (freiwillige Angabe) / Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

Augmented Reality fur die Visualisierung von Windparks mit Hinderniserkennung

Augmented Reality for the Visualization of Wind Farms with Obstacle Detection

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 02.08.2022

Ort, Datum/City, Date — Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 02.08.2022

Ort, Datum/City, Date — Unterschrift/Signature

Contents

1. Introduction

1.1. General Interest

Electricity is one of a country's most important resources, but its sources could not be more different. For decades, energy was obtained only from fossil fuels, which, are finite and thus cannot be intended for long-term use. In addition, the extraction of fossil fuels is a massive interference with nature and leads to a negative impact on the climate [1].

Nuclear energy is one of the cleanest ways to generate energy [2], but the storage, treatment, and decay of radioactive waste is a problem that remains unsolved to this day [3]. In addition, the public image of nuclear power plants has fallen into disrepute at least since the Fukushima super accident in March 2011 [4].

Renewable energies are therefore the only logical way to ensure a secure and long-term energy supply [5]. The share of renewable energy in Germany was 34.9% in March 2019, of which 17.3% stems from wind turbines. Thus, the share of wind energy corresponds to almost half of all energy from renewable sources. In 2020, the energy generated from wind turbines was about 131 terawatt-hours, of which 79% was from onshore wind turbines [6]. As a comparison, the state of North Rhine-Westphalia consumed 114.5 terawatt-hours in total in 2019 [7].

In recent years, politicians have passed more and more laws to expand renewable energy. The current goal is to obtain at least 80% of the energy needed in Germany from renewable sources by the year 2050 [8]. The focus here lies on energy from wind power. The plan to switch to renewable energies and thus protect the climate was again confirmed in 2021 at the UN Climate Change Conference in Glasgow [9]. Furthermore, these plans are currently (as of March 2022) strengthened by the European desire for independence from Russian energy supply [10].

The question of procuring electricity from renewable sources is, thus, an issue which will affect current and future generations. An early expansion of wind power, solar, and thermal plants is therefore indispensable. However, the placement of such plants is not arbitrary. Factors such as proximity to power or residential facilities, existing infrastructure, weather conditions in the region, or political guidelines play a major role. The opinion of the citizens living in the region is also a factor that cannot be disregarded [11]. Even though society's understanding of the need for renewable energy is strong, there is clear difference between society's desire for more energy from renewable sources, also called green energy, and the willingness to live near such a facility [12].

Energy companies and investors are often seen to prioritize profit and their benefit and fail to identify with residents living near wind turbines [13]. Aspects such as noise from rotating blades, shadow impact, and visual pollution of the area are the main reasons given by residents against the construction of wind turbines in their vicinity [13] [14]. Of course, companies can show rendered images of the construction project to the citizens, but the effectiveness of such images is questionable, as they may only show the *best* side of the wind farm, respectively those wind turbines that are farthest

away from nearby houses.

From this thought, the idea of this bachelor thesis was born. The idea is to create a visual support tool that allows citizens to visually and personally experience the planned construction of wind turbines on their cell phones. An app for Android and iOS is to be created that will use augmented reality (AR) to give a better and unclouded impression of the visual impact of wind turbines.

To improve the acceptance of such an app, the challenge of obstacle detection and its management should be implemented. Obstacle detection refers to the acquired knowledge of a program that a real object is in front of another (not necessarily real) object. With this knowledge, the program should then act accordingly and occlude virtual objects by real objects that would occlude this virtual object if it were real. This part is called obstacle occlusion. This results in a virtual wind turbine not being (fully) visible when it is actually located behind real objects, such objects can be trees, larger buildings, or hills, for example.

1.2. Related Work

Facilitating the visualization of planned projects through AR and, therefore, increasing social acceptance is not a new approach. Surveys among Norwegian workers in the architecture, engineering, and construction sector have shown that a visual (digital) presentation of construction projects is on the one hand, very desirable and leads to a better idea of the project, but on the other hand is less used than classical visualization methods such as technical drawings or sketches [15].

While this may be due to the fact that these classic methods are easier and cheaper to design, they are also more appropriate for addressing the corresponding target group. However, with the new generations that have grown up in the digital age and are more critical regarding public promises [16], a shift in the direction of digital visualizations and those in the form of a virtual experience is inevitable. Studies show that conviction (in this case of a construction project) is more likely to be achieved if the affected target group (here the citizens) can make their own experience with the product (here wind turbines in their vicinity) [17].

For a long time, AR was a vision of the future known only from science fiction movies, but at the latest since the release and subsequent worldwide success of Pokemon Go, every young adult who uses a cell phone on a regular basis is no longer unfamiliar with the experience of AR. For months, Pokemon Go has managed to get teens and adults out of the house to use their phones to collect virtually created characters and compete against each other [18]. In the process, reality has been augmented with digital content and made more interesting.

It is precisely this perception that is the decisive factor in convincing users of something new. The best way to convince is to offer the end-user (or in this case the people concerned) the opportunity to actively engage with the matter and thus make their own experience.

The idea of making wind turbines more easily accessible to citizens with the help of digital media is already partially explored. For example, a paper was presented

at the "WindEurope Summit 2016" describing the positive effect of 3D AR on the acceptance of wind turbine construction projects [19]. Here, high-resolution images of the surroundings of the planned turbines were used, and then the wind turbines were inserted into the surroundings. The end-user can then use the platform to move through the area and see the planned wind farm. The approach taken here was to visualize the wind farm on end devices to make it more tangible for neighboring citizens and strengthen approval of its economic benefits. However, it is not clear from the paper whether this approach also addresses the obstacle detection aspect.

In mid-September 2021, Energie Baden-Württemberg AG (EnBW) presented a visualization tool that is intended to provide a realistic representation of planned construction projects and their impact on the landscape at an early stage [20]. However, the aspect of obstacle detection is not mentioned here either.

The question at hand is therefore how good a visualization using AR can be if the aspect of obstacle detection and its corresponding handling is not considered. How is a citizen supposed to imagine the extend of the wind turbine's impact when he points the cell phone at his house and the wind turbine model extends from his first to third floor, although being located behind the house, thus it should not even be recognizable from the user's current position?

Both Apple and Google have been working on a solution for obstacle detection. Any phone that is capable of taking a portrait photo needs some sort of interpretation of depth in the image. However, this doesn't need to be as accurate with longer distances, so that many smartphone software assume that portrait photos could only be taken at a distance of up to two meters. First with the iPhone 12 (Pro version only) did Apple introduce the LiDAR (Light detection and ranging or Light imaging, detection and ranging) scanner. This enables the creation of a more precise depth map of the photo using the phone's wide-angle lens. Google has taken a similar approach, using the image from the main camera and the device's motion sensor to get a more accurate idea of how the user is moving through space to estimate how far away objects are from the viewer [21]. However, Google and Apple's approaches have one major drawback, namely device compatibility. Apple only relies on the top iPhones, (currently the iPhone 12 Pro and iPhone 13 Pro, both the base and so-called "max" versions), and Google has its own requirements for the hardware used in Android smartphones. However, it can generally be said that the upper-class Android phones of the last two years fulfill these requirements [22].

1.3. Contribution

The goal of this work is to find a way to implement the visualization of planned wind farms on mobile devices (both iOS and Android) including obstacle detection and therefore standing out from other AR solutions, by developing a way to increase the social acceptance of planned wind farm constructions. It should be ensured that not only the top-notch smartphones can use this visualization tool.

Many positive effects can be derived from such an app. On the one hand, as already mentioned, the facilitation for the person concerned to imagine a wind farm in his

vicinity, on the other hand, it also facilitates the presentation of wind farm projects to investors for the energy provider companies.

Last but not least, it should be mentioned that this work is not funded or promoted, the app is developed purely with academic research in mind. Therefore, this work can be seen as an independent and uninfluenced achievement, creating a certain basic trust by its users.

1.4. Outline

To create an app that allows users to view planned wind farms on their cell phones in their neighborhood or familiar surroundings, the development process documented below is divided as follows:
In Section 2, the general idea is backed up with a concept, as well as a description of benchmarks, essential requirements and features, that are to be implemented in the app. Furthermore, the actual model is described on a technical basis, later in Section 3 the code for the most important aspects of the app is presented, whereby the reader accompanies the development process. Then, this implementation is tested and the insights, as well as challenges, are documented. The mentioned insights are then grouped into solvable and unsolvable problems, whereas possible solutions are either discussed or the reason why no such solution might exist is explained. Later on, in Section 3.5 the results of the implementation are mentioned and, further on, discussed. Finally, the findings and results are summarized in the conclusion in Section 4 and a possible outlook on further elaboration possibilities is mentioned.

2. Idea Elaboration

In this section of the thesis, the idea of the app is supported by a concept, including aspects of functionality and essential requirements for measuring the success of the app. These aspects are taken up again in Section 4 when comparing the developed app with the stated idea at the beginning. Furthermore, a model, based on the before mentioned idea is developed and described in detail.

2.1. General Concept

The idea behind the app can essentially be divided into two parts. Firstly, the distance between the user, the planned wind turbine, as well as all objects that are in one line with the user and the planned wind turbine must be correctly estimated or measured. Here, obstacles are identified by a distance to the user less than the distance between the user and the wind turbine.

Calculating the distance between the user and the wind turbine should not be a big challenge. Since smartphones have GPS sensors that can be used to determine the exact position of the user, and the position of the planned wind turbine is given the calculation is just the distance between two given coordinates in 2D space (but yet including the curvature of the earth).

On the other hand, calculating the distance between the user's current position and the objects that are in a straight line between the user and the virtual wind turbine, as well as behind the planned wind turbine, is more difficult because the coordinates of these objects are not given. The objects behind the planned wind turbine also play a role, since those are the objects that should be occluded by the virtual wind turbine.

Secondly, the virtual object must be correctly rendered into the image on the user's mobile device. Here, it must be ensured that only the parts of the virtual object that are not obscured by real objects are made visible to the user. This part thus covers the obstacle occlusion aspect.

There are different approaches to determine the distance between the camera of the mobile device, and thus the user, and the real objects that are in line with the user and the virtual object.

One approach would be to determine the size of the real objects, for example by detecting and classifying the objects and storing standard sizes for certain objects, which are then compared with the actual pixels used on the sensor. By relating the size of the object in real life, as well as on the sensor, and the focal length of the camera's sensor, the distance between the camera and the object can then be determined. Even if the focal length cannot be read from a so-called "ImageStream" on either iOS or Android, this problem could be circumvented by taking a photo to retrieve the metadata before starting the image stream, which is later needed to place live virtual objects. Also, the user could be limited so that only the main camera can be used for AR resulting in no need to consider different focal lengths of sensors on phones with multiple cameras. However, this approach has a significant drawback. For one, the objects need to be correctly identified, this could work via a neural network for example, but this iden-

tification can be resource and time-intensive, hence performance would be affected. On the other hand, the bigger problem is that it is not only necessary to identify the object correctly, but also how much of that object can be seen. For example, a house may be partially obscured by another obstacle, in which case the calculation of the distance to the house would have to be adjusted to the visible part only. In other words, this approach can only work if the identified objects are always fully visible and not themselves obscured. Since this is very intensive image analysis, this approach is discarded.

Another approach would be not to have to identify the objects via a neural network but to have a map on which objects such as houses, or forests are noted. With the help of the direction of the cellphone's camera and the underlying map, the visible objects can be identified. With the stored heights of, for example houses or trees, the above-mentioned calculation can be made. However, the problem remains that objects must not be partially hidden, otherwise, the calculation will fail again. In addition, the underlying map must always be kept up to date, should a piece of forest be cut down to build family houses, this may drastically change the expected height of the visible object and the distance would be calculated incorrectly. Thus, this approach is not an acceptable solution either.

Yet another approach would be to use the digital terrain model offered by the state of North Rhine-Westphalia [23]. Here, the height of the terrain was measured and publicized as a grid. The size of the grid can even be set to as accurate as 1m x 1m. This data has the advantage that one can put an invisible layer of the heights from the grid over the terrain. Therefore, no object detection nor calculation has to follow. Then only the distance to the wind turbine must be calculated, whereby then the size of the wind turbine can be determined in pixels and then rendered into the image. The size of real objects has become uninteresting since the calculated layer covers them. However, calculating this layer is an impossible task for a smartphone. A 1m x 1m grid is three megabytes in size [23]. The memory storage the smartphone would have to provide would be unfeasible with today's technology. If one decides to use a larger grid, then the file size is reduced proportionally, but at the latest when rendering the wind turbine, the problem occurs that it might appear floating if the grid would not have the registered height on the entire surface. Because of performance reasons and limited memory on the phone, this approach is discarded.

Thus, a suitable solution should not depend on real objects being fully visible on the image, nor is it guaranteed that a stored value for an objects height will apply to every instance of such object. A possible solution would be to use depth analysis of an image. Both Apple and Google provide such functionalities in their respective AR/VR toolkit. Here, all that would be needed is to analyze the image obtained and estimate the depth of each pixel. The size of the wind turbine can then be calculated using the distance between the user and the wind turbine. If the pixels in which the object is to be rendered do not have at least the same depth as the distance to the wind turbine, the app can be sure that a real object occludes the wind turbine.

After that, rendering the virtual objects is just a matter of creating layers of the image. Here, the image can be divided into two layers, one layer containing all pixels

of the image that are closer to the user than the wind turbine, and a second layer containing all pixels that are located behind the virtual object. After that, the wind turbine must be inserted between the two created layers as an additional layer. This guarantees that all objects in front of the wind turbine occlude the corresponding parts of the wind turbine while, at the same time, the corresponding areas of the background are occluded by the virtual object. Here, it is assumed that the wind turbine is only defined by its position and has no depth of its own. The fact that a rotor blade can be further away from the user than another blade of the same wind turbine is neglected.

2.2. Definition of Benchmarks

After the general idea of the app has been described, benchmarks must now be defined for the app, with the help of which the success of the app can later be evaluated. The benchmarks can be divided into ”essential requirements” and ”features”. Essential requirements represent the technical challenges that are to be overcome. Features, on the other hand, represent the additional functions of the app. Those features can be further divided into essentials and gimmicks. While essential features are required for the correct behavior of the app, gimmicks are additional features that only extend the basic function of the app, but their absence does not affect the correct behavior of the app.

2.2.1. Essential Requirements

The app shall be designed for mobile devices. This also includes the consideration of the limited computing power of mobile devices. Specifically, for this bachelor thesis, it is intended that the app works on smartphones with both Android and iOS operating systems. The augmented wind turbines should be displayed at the correct location (indicated by the longitude and latitude of their position). For this, it is crucial that the user's current position is also considered. Furthermore, only those virtual objects currently visible by the camera should be visible on the screen, this also includes that the parts of the virtual objects that are occluded by real obstacles are not to be presented on the screen (obstacle occlusion). For this to work the obstacle detection has to be handled correctly. Another criterion is the possibility to use the app without internet access. Since wind turbines are not built in densely populated areas, and the network coverage of the mobile internet is often not good, one of the most important aspects is that the app can be run completely or at least to a large extent on the mobile device alone. Finally, it is also important for this bachelor thesis that the created app is written with the JavaScript programming language since the augmentation part will eventually become part of an already existing larger Ionic Vue app.

2.2.2. Features

The essential features include the use of the main camera of the mobile device, as well as the correct use and reading of the sensor data provided by the smartphone, such as

GPS and gyroscope. Another important feature besides the correct determination of the user's position is the correct positioning of the virtual objects. For this purpose, the choice and settings of the rendering library are of great importance. Moreover, even though the occlusion is already an essential requirement, it should not only function but also improve the user experience. Therefore, ideally, the app works smoothly on the majority of devices, which means that the image is augmented with at least 24 frames per second. However, this depends on the implementation of the chosen solution and the computing power of the device. This is an important feature, but not one that can be guaranteed.

Non-essential features include, for example, info boxes that contain additional information about the wind turbine and are only displayed when the wind turbine is clicked. In addition, a compass in the form of a north-south needle could also be displayed. This would resemble Google's well-known map app and possibly contribute to user acceptance. Also, the integration of a screenshot functionality that allows the user to show the planned project to friends and family later on, could improve the user experience. Additionally, a part of a map could be displayed so that a user does not have to switch back and forth between a map app and the AR app to locate himself. Arrows on the left and right sides of the screen that pop up when the camera is not facing the wind turbine would be another possible additional feature, which could make the app easier to use. Other gimmicks, such as a virtual hot air balloon, would also be conceivable, which do not add anything to the functionality of the app, but show what is possible with AR.

2.3. Selection of Core Technology

In the following subsections of the bachelor thesis, the question of what the most suitable technology is to implement the idea of this thesis is addressed. The initial planning represents an important milestone. Here, the criteria under which technology should be selected must be carefully considered, the most important criteria were already mentioned in Section 2.2.1.

	ARCore	ARKit	WebXR	AR.js	Unity	ML	Stream
Android	✓	✗	✓	✓	✓	✓	✓
iOS	✗	✓	✗	✓	✓	✓	✓
JavaScript	✗	✗	✓	✓	✗	✗	✓
Detection	✓	✓	*?*	✗	✓	✓	✓
Offline	✓	✓	*?*	✓	✓	✓	✗
Maintainability	✗	✗	✓	✓	✗	✓	✓

✓Criteria fulfilled, ✗Criteria not fulfilled, *?* No information

Table 1. Comparison of possible core technologies in dependency of main criteria.

In the table above, different possible core technologies (visible as columns of Table 1) have been evaluated against the most important criteria (listed in the rows of Table 1).

In general, obstacle detection and occlusion is possible both with ARCore[1] and ARKit[2], but out-of-the-box only within a short distance of up to eight meter. Even though a solution written with ARCore ARKit would not be created directly with JavaScript, it should be possible to have the created apps flow into the Ionic app afterward. However, it results in double the work, since ARCore from Google supports Android and iOS, yet the functionality for iOS is very limited [22]. ARKit is developed by Apple exclusively for iOS, so a double workload is unavoidable. However, solutions written in the ARCore and ARKit frameworks fulfill another important aspect, the ability to function completely offline. Nonetheless, the maintainability is not given here, since two different programming languages are needed to keep the two apps up to date.

WebXR[3] is a project of "The Immersive Web Working Group/Community Group" in collaboration with Google, Mozilla, and Facebook, which is intended to present virtual content to the user in a web solution. XR is the so-called "mixed reality", a mixture of virtual and augmented reality. Among other things, in WebXR, ARCore is used to display AR content in the Google Chrome browser on Android devices. However, a possible solution with WebXR does not meet all the mentioned criteria. Even though it should be a good solution for Android, Apple does not provide the function of XR in its in-house browser. The Google Chrome browser or Mozilla Firefox also do not provide the XR functionality on iOS. Only a special XR browser from Mozilla[4] is supposed to fulfill the properties to display XR content. During testing, however, the app crashed directly when starting the example XR content. Also, the app was last updated in May 2020, and no information on Mozilla's official page for this project indicates that it is actively maintained [24]. Another aspect that is not apparent is the possible integration into an already existing app. Even though the project can be used with HTML and JavaScript, WebXR's documentation indicates that it cannot be part of an Ionic app. This is also the reason why there is no information about the offline capabilities of WebXR.

The fourth possible solution is AR.js[5]. AR.js is a widely used library for providing AR features to the user of websites. Here, AR.js stands out because it works not only marker-based but also location-based. In the marker-based method, a special pattern must be printed out, which is then recognized by the camera and the virtual object can be positioned on top of the marker, while the location-based method requires the exact coordinates of the object to be placed. AR.js fulfills almost all the mentioned important criteria from Table 1. However, AR.js only deals with the rendering of objects, but not whether these objects are partially or even completely obscured by real objects.

Unity[6] looks like an ideal solution at first sight since almost all criteria could be

[1]https://developers.google.com/ar/develop
[2]https://developer.apple.com/augmented-reality/arkit/
[3]https://developers.google.com/ar/develop/webxr
[4]https://apps.apple.com/us/app/webxr-viewer/id1295998056
[5]https://ar-js-org.github.io/AR.js-Docs/
[6]https://unity.com/unity/features/ar

provided with a checkmark. However, there are two problems here, one of which is not obvious from Table 1. On the one hand, a Unity app is written directly in the Unity development environment and cannot be included as part of another app. Thus, it would have to exist as a standalone app. On the other hand, and much graver is that Unity cannot access the graphics chip on Android devices. From another student project at RWTH Aachen, it is known that the AR of wind farms works on iOS devices with Unity but takes too long on mobile devices with the Android operating system.

Machine learning (ML) is designed to deliver an approach to previously unsolvable problems. Machine learning involves training a so-called model (also known as a neural network) based on extensive training and validation sets. With great computing power (usually executed on a graphics card), up to several thousand parameters are then carefully modified so that the model can make predictions that are as close as possible to the actual values of the validation set. It should be noted that a model can only be as good as the data with which it was trained. If the underlying data is faulty or incomplete, this will severely affect the model's performance. With a neural network, different tasks can be mastered, for example there are known networks that can classify objects correctly. These networks are exported to Android or iOS after successful training and can then be used in an app. A neural network that can estimate the depth from an image is thus imaginable if sufficient training data is available. Most neural networks (or better said the algorithm how they are trained) are written in Python on the computer. This solution seems very promising, as here many of the important aspects are provided with a checkmark as well. However, the following two questions arise: First, the availability of correct and meaningful data sets to train and validate the model. Second, it is unclear whether the computing power from the mobile devices is sufficient to apply the trained network efficiently.

Finally, the possible solution of a video stream means that the mobile device streams the camera image to a server, which then sends the rendered image back to the user. The front end of such an app could be written in JavaScript here and thus be easily integrated into the app. There should also be no problem with the different operating systems, as it would be a typical video stream. With an improved processing power of the server, obstacle detection would still be a challenge, but more manageable than on a mobile device. However, the offline availability aspect is not a given with such a solution. Even with approaches to upload only parts of the captured image, latency would still be a strongly impairing factor in the use of the app. Poor network coverage in the region of planned wind farms would even prevent the app from functioning correctly. Thus, while this is a possible idea for areas with great network coverage, it is not a valid approach to solving the problem.

As in most cases, there is no ideal out-of-the-box solution but a combination of different approaches to a possible overall solution seems to be the goal here. For the previously mentioned reasons, the possible solutions with ARCore, ARKit, WebXR, Unity, and the video stream must be excluded. Each of these solutions has negative aspects that cannot be overcome, even in combination with other solutions. However, a combination of machine learning and AR.js is conceivable. In this case, a neural network would have to be trained on the computer and later integrated into an app

on the mobile device. The focus here is on a neural network that determines the absolute distances for each pixel in a given image. The distance of the pixels from the mobile device can then be used to create what is known as a depth map. With the information from this map, both the depth map and the original image can be divided into different areas. It would make sense here to divide the image into sub-parts ”in front of the wind turbine” and ”behind the wind turbine” so that the virtual object can be placed between these two parts as yet another layer.
The workflow would look as follows:

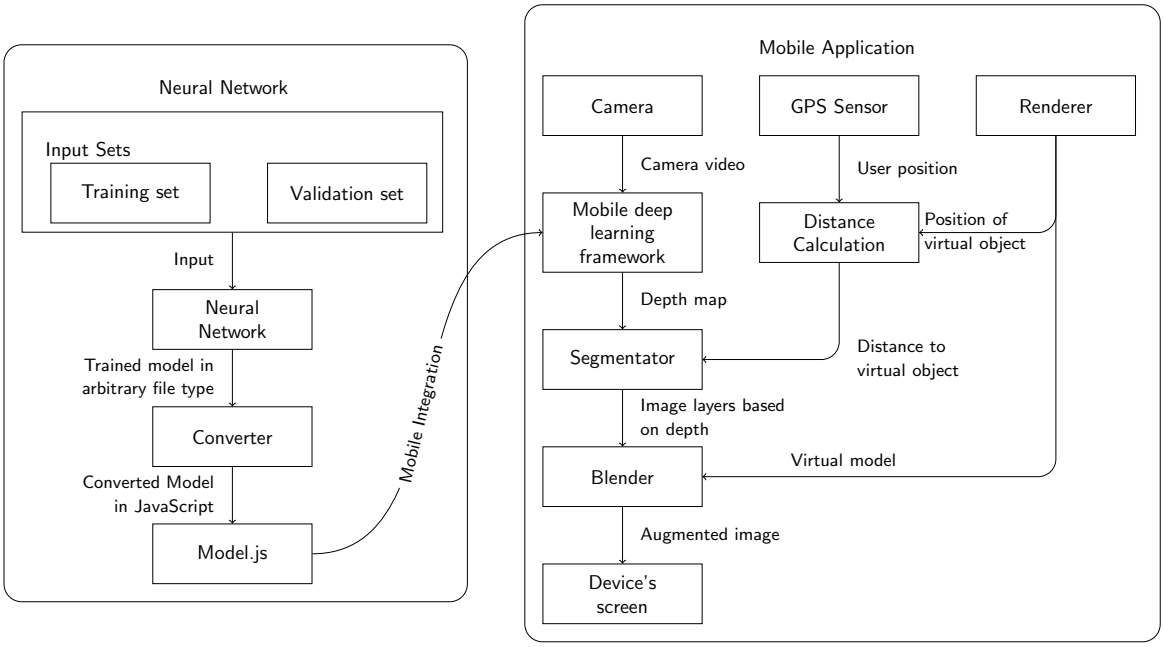

Figure 1. General workflow of concept divided into work to be done on the computer (left) and work to be done on within the mobile application (right).

As can be seen in the diagram, the workload is divided into two areas. Firstly, the neural network has to be prepared on a computer, and secondly, all following time and location specific tasks have to happen on the mobile device.

2.4. Neural Network

As can be seen in Figure 1, the task on the computer is to train a suitable neural network and later convert it to a model type that can be executed on mobile devices. The most important task here is to find and select suitable training and validation sets. Of course, it is not said that creating a model is an easy task, but for an expert in this field, obtaining valid sets is the bigger and more time-consuming challenge. Creating own suitable sets goes beyond the scope of the context of this bachelor thesis, thus is not a possible solution.

2.4.1. Choice of Training Data

Finding a suitable set of outdoor images with sufficiently complete data on the depth of each pixel has proven to be extremely difficult. Two comprehensive sets are "DIODE: A Dense Indoor and Outdoor DEpth Dataset" [25] and "The KITTI Vision Benchmark Suite - Depth Prediction Evaluation" [26]. However, both sets have certain advantages and disadvantages. The set from KITTI only measures depths up to 80 meters. If it is not the users' desire to stand less than 80 meters in front of the wind turbine and view it from up close, this range is for this specific purpose too short. However, there are pre-trained models for this set, including their corresponding code. The dataset from DIODE provides better training data in this respect, as the depth of the images is also greater than 80 meters. The sensor used to measure the depth of the images can detect depths up to 350 meters. However, the data is not always complete, and no predefined or pre-trained models are publicly available.

There are other publicly available and detailed training sets such as the "CITYSCAPES DATASET" [27]. However, this is rather intended for training neural networks for autonomous driving than landscape depth calculation. The sets thus consist of various scenes from road traffic whereby important elements of the images have been classified and identified. This set is not suitable for depth computation from a single image.

2.4.2. Evaluation of Neural Networks and their Models

The creation of a neural network is the next logical step once the choice of training and validation data is determined. However, creating a neural network is beyond the scope of this bachelor thesis. Thus, already existing networks and their models had to be used. To understand better the terminology, the difference between a *neural network* and a *model* is that the neural network is the mathematical and programmed description of the network, whereas the model is a trained version of this network. The model is not further refined and can be applied to new inputs (here images), this step is called *inference* [28].

The choice of a neural network and the associated model depended heavily on their availability. When choosing the network, DORN [29], BTS [30], as well as the GCNDepth [31] network stood out. DORN and BTS were both trained exclusively on the KITTI dataset and are capable of estimating distances up to 80 meters. On the DORN's official GitHub page, a predefined model trained on the KITTI dataset is available for download. DORN is based on the Caffe [32] open-source framework for deep learning which is developed by Berkeley AI Research (BAIR), The Berkeley Vision and Learning Center (BVLC), and community contributors. The last published version of Caffe is from April 18, 2017, and is consequently no longer state-of-the-art, with just under 1,000 open issues and no adaptation to the latest graphics cards.

BTS on the other hand has been implemented in a framework called "Tensorflow". Tensorflow is an open-source machine learning software library developed by Google.

With over two thousand open issues on the official GitHub page[7], this framework is not bug-free either, but updates and patches are released almost monthly.

Those two networks conform to the two best-ranked, which also include a link to their code on KITTI's benchmark page[8]. It should be noted that the values of the "Scale invariant logarithmic error" (SILog), determining the ranking of the two networks, do not differ much (BTS 11.67, DORN 11.77), but the specified runtime is very different. Both times measured on a GPU with 2.5Ghz, BTS is almost ten times faster than DORN (BTS 0.06s, DORN 0.5s).

In contrast, the third mentioned alternative, GCNDepth, is much more recent. The accompanying paper to the network was not published until the end of 2021. This neural network is written entirely in Python and the official GitHub page also has a pre-trained model available for download. The aforementioned network is built on the open-source PyTorch framework, which is mainly developed by Facebook's AI Research lab (FAIR). By its own claims, the GCNDepth network has a SILog of 15.54, but no indication of the duration of the inference was provided.

All of the networks mentioned claim to have made a great advance in the determination of depths compared to previous methods. Therefore, it is not yet possible to weigh which of the networks will provide the greatest advantage in implementation. It will be part of the later evaluation to investigate this matter in more detail.

2.4.3. Conversion of Model

A neural network model that has already been created must first be converted for use on a mobile device. This conversion performs an adaptation to the framework available on mobile devices. Even though well-known frameworks such as Tensorflow have their own framework specifically for mobile devices, this is not necessarily the case for all of them. It is also worth saying that conversion can affect the performance of the network. By having to recreate the network in the definitions of a different framework, as it was created in, any performance adjustments will possibly not be applied correctly.

In the field of mobile machine learning frameworks, Tensorflow was the market leader until last year, currently PyTorch is just as popular [33]. However, other formats also offer efficient mobile solutions. For example, ONNX offers a mobile JavaScript framework for networks in .onnx format. ONNX stands for "Open Neural Network Exchange" and represents a file format that can be opened and edited in various frameworks [34]. The determining factor for the performance of a network is the time of inference. Here ONNX.js claims to have a clear advantage over other frameworks such as TensorFlow.js and Keras.js, yet another deep learning framework [35]. Nevertheless, it is impossible to say whether the accuracy of the network may not be reduced when converting from another file format to .onnx.

Each file format must ultimately be converted differently. Some converters require the weights of individual so-called neurons in addition to the model to save a model in a different format, others only require a special command in the command

[7] https://github.com/tensorflow/tensorflow

[8] http://www.cvlibs.net/datasets/kitti/eval_depth.php?benchmark=depth_prediction

line [36] [37] [38]. Since all mobile frameworks claim to work well, no preferred choice can be made so far. It will be part of the testing to compare the performance of these frameworks.

2.5. Mobile Application

Most of the work is to be done on mobile devices. Here, a ”mobile device” usually means a smartphone, since a large part of the population owns one and carries it with them constantly. Nevertheless, a mobile device can also be a tablet or a portable computer. The software on the mobile device is tasked with getting the camera image and analyze each frame with the help of the neural network. This analysis results in a so-called depth map. The depth map contains an estimated distance per pixel of the image. At the same time, the distance of the user to the virtual objects to be placed is to be calculated. Using this distance and the depth map, layers can be determined from the camera image with the help of the segmentator. A layer is an image with the same dimensions as the input image, but only the pixels that fulfill a condition are displayed here, all other pixels are transparent. The condition, in this case, is whether the distance between the user and the object is greater than the depth of the pixel, in other words, whether this pixel should be in the foreground. The layers created here can then be extended by the virtual object, created by the renderer, placing the object between the layers with the help of the blender. The output of the blender should then be an image with the same dimensions as the initial input image and output to the user on the screen.

2.5.1. Choice of Renderer

The renderer takes over the task of placing and displaying virtual objects in the app. In general, AR can be divided into three different functionalities: marker-based AR, marker-less AR, and location-based AR. Marker-based AR works in such a way that the camera has to recognize a so-called ”marker”. A marker can be a QR code or a previously defined image, which must then be printed out or displayed on a screen and the camera of the cell phone has to be pointed at it. Should a marker be used, the virtual object can be guaranteed to be in the correct location all the time since it re-positions itself on the marker in each frame, therefore, this approach provides great stability to the augmentation. A major drawback of this approach is that the marker must always be visible to the camera. Ideally, the marker should be orthogonal to the camera, since any shift or different angle must first be compensated for. This approach is great for small AR projects to display on your desktop. However, this approach is unsuitable for the implementation of this app because either no marker could be large enough for the camera to detect it from a greater distance, or the angle would be too narrow to identify the marker correctly.

The second method of displaying virtual objects is marker-less ar. With the help of the camera, the app scans the surroundings of the cell phone and tries to determine which elements of the image represent a surface to be able to place virtual objects.

Different textures of the environment are one of the main clues for the software. Modern mobile devices also have advanced sensors, such as a laser scanner or a LiDaR sensor, that help determine these surfaces. Since the texture of a surface can no longer be determined accurately beyond a certain distance, the mobile device can also no longer decide this with certainty. Even though this distance depends on the technology used as well as the resolution of the camera, it is safe to say that it is less than the distance from which a user wants to view virtual wind turbines.

Location-based AR is the third option for rendering virtual objects on the camera image. In this method, both the GPS sensor and the gyroscope of the mobile device are addressed, and each virtual object to be rendered must also be provided with GPS coordinates. The GPS sensor is then used to determine how far the user is located from the virtual object, this assists in calculating the size of the virtual object. The gyroscope sensor is used to determine if the camera is facing the direction of the virtual object, only if this is the case the object should be visible. Since this possibility is independent of the quality of the camera, the presence of other sensors like laser or LiDaR, as well as the distance to the virtual objects, is the most suitable variant to output these objects on the screen. In the research, AR.js turned out to be the most promising renderer. AR.js is built on top of AFrame, which in turn is a simplified API for Three.js. Three.js allows the creation of virtual objects, which are then used by Aframe to combine and place them, AR.js also uses the camera and other sensors of the cell phone and can therefore offer both a marker-based and a location-based solution.

An alternative to AR.js would be argon.js, for example. argon.js is just like AR.js a library to connect virtual 3D objects and 2D content with the user's real-world image. Here, three important components of the library are used to present the different inputs to the user. The three components are the Reality Manager, Reality View, and Reality Augmentator. The Reality Manager is responsible for distributing the data from the smartphone (sensor, tracker, user input) to the Reality View and Reality Augmentator. Furthermore, the Reality Manager has to manage the virtual objects and their current state and finally render the user interface on the screen. The Reality View receives the user input and the state of the virtual objects from the Reality Manager, updates the state of these objects, and returns the new state to the Reality Manager. The Reality Augmentator has the task of presenting an augmented version of reality using the virtual objects and their state from the Reality Manager as well as the Reality View.

However, based on the fact that the latest published version is just under 6 years old [39], parts of the API documentation are still marked as "TODO" [40], and there are no compelling examples online of the usage of argon.js, this approach is refrained from pursuing.

2.5.2. Choice of Segmentator

In the search for a way to segment an image based on a condition, many other neural networks were found. However, these segmentators categorize the image based on its

content and not on a pre-existing depth map. The manual approach to this problem is to handle the pixel values from the provided image one by one. Typically, for a low-level pixel-wise calculation the widely spread library "OpenCV"[9] is used. However, the documentation suggests no function nor implementation to use a depth map to modify images.

Since the video image from AR.js is presented to the user in a HMTL canvas element, its context can be determined, and the image data extracted. Each pixel can then be assigned to either the foreground or background based on the value from the depth map and the distance to the virtual object. To simplify the work, the depth map can first be modified with the help of the user's distance to the wind turbine. The depth map can be in different formats, for example, a two-dimensional array. However, since the extracted image data of a canvas element is represented in a one-dimensional array, and the conversion from a two-dimensional array to a one-dimensional array is not challenging, the simplifying assumption was made that the depth map is also represented as a one-dimensional array. Using the distance of the user to the wind turbine, a new value can now be assigned to each value of the depth map, for example, $true$ should this pixel belong to the foreground, otherwise $false$. Values like 1 and 0 would also be conceivable. Since JavaScript is dynamically typed, it has no negative impact to change a value of type number to a boolean if necessary. To avoid confusion, a new array can be created with those values, in the following referred to as *binary map* (BM). This results in the following formula:
Let I be the total number of wind turbines, D be the depth map with ℓ entries and d_i be the distance of the user to the wind turbine i, then

$$BM_{i,j} = \begin{cases} true, \text{if } D_j < d_i \\ false, \text{ otherwise} \end{cases}, \ 0 \leq i < |I|, \ 0 \leq j < \ell.$$

To determine the foreground and background of the image, a for-loop can be used to split all pixels of the initial image element into newly created canvas elements. All pixels that have the value $true$ at their corresponding position in the depth map can be painted on the foreground canvas element, the other pixels onto the background. Pixels that should not be drawn into the respective element must be replaced by an arbitrary pixel with 100% transparency.

Let FC_i be the foreground canvas element for the wind turbine i and BC the background canvas element, BM_i the binary map for a wind turbine i, T the one-dimensional array containing all pixels of the image, and p^* an arbitrary pixel with 100% transparency:

$$FC_{i,j} = \begin{cases} T_j, \text{ if } BM_{i,j} = true \\ p^*, \text{ otherwise} \end{cases}, \ 0 \leq j < |T|.$$

$$BC_j = \begin{cases} true, \text{ if } FC_{i,j} = false \text{ for all } i \\ false, \text{ otherwise} \end{cases}$$

[9] https://opencv.org/

With this approach, the different images represented as canvas elements can be created, which then have to be merged by the blender. This completes the part of choosing the way to segment the image.

2.5.3. Choice of Blender

The blender has the task of merging several layers into one image.

The library "context-blender" by Gavin Kistner[10] has not been updated for more than seven years, yet is referred to in many image-blending processes. With the help of this library, a canvas element can be described with two further canvas elements, a blending mode, and further options. Here, the element passed first represents the foreground, and the second element is the background of the final image. However, the content of the foreground is only merged with the content of the background based on the mode. If the background is transparent, for example, the colors will be softened or darkened. Therefore, this library is not an option for the planned project.

The library "merge-images" by Lukas Childs[11] is popular, with more than 100 thousand downloads per month. Using this library, multiple images of a file format can be layered on top of each other. Other options can be applied, such as a shift on the X or Y axis, respectively, as well as the degree of transparency of each layer can be adjusted. The result of this function is an image in .PNG format, which can then be inserted into the document object model (DOM) in a *img*-element as *src*-property. This library is promising, but strongly dependent on the file format of the individual layers after segmentation.

Another way to layer different images that does not require an external library is to use pure CSS. HTML elements can be positioned pixel-precise on the screen with the help of the CSS attribute "position: absolute;". Different elements can be placed on top of each other by using different values in their CSS attribute "z-index". Combining these two attributes does not create a final image element, but instead keeps different layers. However, since parts of the foreground are transparent, this procedure cannot be distinguished from creating a single image. One advantage of this method is that it is not necessary to create a element to display the background. Since the video is located below all other elements in the DOM, it is sufficient to display the foreground in a canvas and place it over the video.

In summary, it can be said that the manual approach is preferred due to the special use case and the lack of suitable libraries. This applies to the calculation of the foreground as well as the correct positioning and output on the screen.

[10] https://github.com/Phrogz/context-blender
[11] https://github.com/lukechilds/merge-images

3. Implementation and Insights

In this section, the implementation of the model defined in Subsection 2.2.1 is addressed. The technical details of the used computer are as follows: Processor: 11th Gen Intel(R) Core(TM) i9-11900K @3.50GHz 3.50GHz, Ram: 32GB, GPU: NVIDIA GeForce RTX 3070 Ti, OS: Windows 11 64 Bit, Ubuntu 21.04 64 Bit. Software versions are as follows: node v17.4.0, ionic 6.18.1, npm 8.5.1, Chrome 99. Testing device: OnePlus 6 with Android 11.

3.1. Implementation of Neural Network

The neural networks mentioned in Section 2.4.2 presented some unexpected difficulties. The DORN network could not be reproduced because the installation of Caffe was unsuccessful. This had several reasons, the libraries needed to install Caffe are partly outdated or do not work one the latest operating system, neither Windows nor Ubuntu. Caffe is not explicitly needed when translating the model with a converter to another file format, but in converters like "caffemodel2pytorch" or "Caffe2Pytorch" the file containing the weights of neurons is needed besides the .caffemodel file. These weights are not available for download on GitHub. An email to the owner of the repository remained unanswered.

The BTS network could not be reproduced also by installation difficulties and afterward problems with the execution. A conversion into another file format was omitted due to the fact that the determined distances would not represent absolute values.

This has also been the problem with Intel's "MiDaS" neural network. Even if this network was not trained exclusively on images and depths in landscapes, it appeared to be a suitable network for this task. Positive aspects of this network are easy usability and the wide range of training data. In addition to the images of KITTI, a total of ten different data sets were used including 3D movies [41]. As another aspect, the provision of a smaller network, explicitly for use on mobile devices also was in favor for the network. After email consultation from René Ranftl, one of the authors of the paper and network, it became clear that the relative values of the depth map cannot be easily converted into absolute values. To illustrate, relative values simply describe the relation of pixels to the user. Lower values express closeness to the viewer, thus larger values express greater distance. However, these values are not related to the actual absolute distances between the camera and the object.

Through the mentioned consultation it became apparent that a conversion is only possible if in each image with which the network was trained and in each image on which the network is applied, there are at least two pixels each to which the absolute distance is already known. Since this is not the case and cannot be guaranteed, this network was also discarded. Nevertheless, this insight was groundbreaking for the further course of the search for a suitable network.

The GCNDepth network was then examined more closely, yet, despite the clues of the authors, this network is also not capable of evaluating absolute distances. It also calculates only relative distances. The same authors published the paper "Absolute

distance prediction based on deep learning object detection and monocular depth estimation models" [42]. Here, the network described makes use of two underlying neural networks, including the GCNDepth network. Using a classification of objects, estimating how big these objects are, and then comparing those values with the relative distances given by the depth map, absolute values are then determined. These values are quite accurate according to their own data.

However, the challenge arises here that not all objects are recognized or classified. The underlying classification classes are defined by the COCO network [43]. Also, unfortunately, no distance is explicitly specified here on which the network works reliably. Nevertheless, since the GCNDepth network was tested on KITTI, at least 80 meters can be assumed. However, the problem of the classification and the estimation of the size is the same as already described in Section 2.1, the estimated objects must be completely visible, otherwise, the estimation is biased. However, the biggest problem was that no model to the network described in this paper was made available for download and hence is not usable for this work.

As a consequence of the reasons mentioned above, the search for a neural network was discontinued. Nevertheless, in order to adapt the app to the use of a neural network, a black box was built to simulate the existence and use of a network. This black box is further described in Section 3.3. Omitting the neural network in the current implementation nevertheless allows the definition of performance and behavioral requirements of such in the remainder of this thesis. This characterization is described in more detail in Section 4.1.

3.2. Integration of AR.js and Wind Turbine Model

AR.js is one of the most popular open-source libraries for augmented reality. To use it on a website, both a version of A-Frame and a version of Three.js must be included as scripts in the *head* of the HTML page. How to include the scripts can be found in Appendix A.

The included scripts may load additional data in the background, for which an Internet connection is mandatory. After these two scripts are included, a new *a-scene*-tag can be created in the *body* of the web page. This is a native element of A-Frame and is needed to manage all the underlying functions of Three.js. Also, the *a-scene* will include all the virtual objects to be displayed. It is important to know that there can be only one such element per page. A UI element to go into virtual reality mode can be hidden by setting the "vr-mode-ui" attribute to "enabled: false". By setting the "arjs" attribute in the *a-scene* the functionality of AR.js will be included in the scene.

Each scene needs a *a-camera* element. This element is responsible for controlling the virtual camera in A-Frame. However, since the camera from the cell phone is to be used here, most of the settings are made in the AR.js-specific attribute "gps-camera". With the help of the values of "gpsTimeInterval" the constant request of the current position can be delayed by the passed value. "minDistance" as well as "maxDistance" set a frame in which distance virtual objects should be displayed on the screen. The "rotation-reader" tag allows AR.js to get access to the orientation of

the camera and the tilt of the phone. Finally, the "arjs-look-controls" attribute was added to avoid arbitrary jumping around of virtual objects, more about this in Section 3.7.1. All virtual objects must now be included before the *a-camera* element within the scene. This can be done either manually, or dynamically via JavaScript. Here the dynamic approach was chosen, because it simplifies the further development, for example with the help of a database, which provides the information about the wind turbines. Implementation can be found in Appendix A.

However, it should be noted that an *a-scene* must be a direct child of the body element. Since this cannot be guaranteed by the Vue framework, the entire AR part of the work was wrapped inside an *iframe* and then embedded in the Vue app.

To dynamically create wind turbines the possibility to create elements via components provided by AFrame was used. The implementation of the component registration can be found in Appendix A. For the visualization of the wind turbine, a *.gltf* file was used. glTF stands for "Graphics Language Transmission Format" and is a standardized file format for three-dimensional models [44]. The component can now be used to set values for the attributes "speed" (rotational speed of the wind turbine), "heigtScale" (height of the wind turbine), "rotorScale" (length of the wind turbine blades), and "direction" (orientation of the wind turbine). For this purpose, an *a-entity* element is created, the coordinates of the object are assigned as *latitude* and *longitude* values to the "gps-entity-place" attribute and then the wind turbine attribute is added with the previously mentioned values. Another model was created to handle toggling the rotation of the wind turbine, which will be discussed in Section 3.7.2.

3.3. Flow of Program

This part of the paper describes the main skeleton of the app and goes into detail about the individual components as described in Section 2.5. Some parts of the implementation happen sequentially and the output of single functions is used as input for the following functions. To avoid constant allocations of memory, data that must be used by multiple functions and may not change in each iteration have been defined as global variables. It is important to note that the functions described below must all be included within the iframe that represents the augmented reality component of the app.

The implementation is divided into two main parts, the initialization, and a loop. In the initialization, the app first waits for the video element of AR.js to load. Loading the video completely is important because the video can be larger than the pixels that are output to the screen. To ensure that the video is centered, AR.js adjusts the CSS attributes of the position. This position is then determined and stored by the function *findVideoPosition()*. Since the position of the video will not change after being initialized, this function only needs to be executed once. As another part of the initialization, for example, the neural network can be loaded and instantiated at this point.

After this the wind turbines must be loaded with the help of the function *loadWindTurbines()*. The number of turbines to be displayed is also determined in this way.

Here the turbines are loaded from a JSON file and the corresponding *a-entity* elements are created. Since the loading of the wind turbines is asynchronous, the number of wind turbines to be displayed was also specified manually.

With the help of the correct position of the video and the amount of wind turbines to be rendered, the overlay canvas elements can then be created and placed correctly, matching the device's screen. Since there is one overlay canvas element for each wind turbine, representing the foreground which lays in front said wind turbine, and only the content of this canvas element is to be constantly overwritten, this also only needs to be created once. At the same time, another canvas element is created that matches the dimensions of the video element. This canvas element reflects the content of the AR.js video, after which it can be limited to only the visible part of the screen. It solemnly serves as an aid for the further calculations but is never displayed to the user, therefore the name *hiddenCanvas*.

Since the correct placement of the overlay canvas elements is crucial for the further course of the app to work, the loop is started at the end of this function, which is also the reason why the wind turbines must be loaded before placing the overlay.

The loop (referred to in the implementation as a function fittingly named *loop()*) is executed only if the boolean *videoRendering* is set to *true*. This attribute is toggled by a button on the HomePage. This design decision will be discussed further in Section 3.7.2. Within the loop, first, the contents of the context of the *hiddenCanvas* are cleared and overwritten with the current video image provided by AR.js. Then the function *trimHiddenCanvasData* is called. This function takes three parameters, the image data of the *hiddenCanvas* just described and the width and height of the AR.js video. The function is responsible for creating new image data, limited to the pixels that are visible to the user. This is partly because applying the neural network to small image data is more efficient, but also because if the image data is too large, the content cannot be displayed correctly to the user. To accomplish this task, the function determines the excess pixels, which is given by $totalOffset = videoWidth - screenWidth$. Since the overlay canvas is supposed to be centered, and it may be that the excess is odd, it was decided that the left offset would be described by $offsetLeft = \lfloor totalOffset/2 \rfloor$ and the right offset by $offsetRight = \lceil totalOffset/2 \rceil$. If there is a difference between the width of the video and the screen, a double for-loop will keep only the value of the respective pixels of the *hiddenCanvas* that can also be displayed on the screen. Afterward, this function returns the pixels both in the form of an array and as an image object, and they are stored together in a single object.

The image data can now be used to create the depth map. This is the part where the neural network would be applied to the image data. Using the current implementation, the neural network can either accept an image (i.e., an image object), or a one-dimensional clamped array with an 8-bit representation of each pixel value (red, blue, green and alpha) of the image. Since, as described in Section 3.1, no neural network was possible to include in the implementation, only the possible input and the expected output of the function are described here. Nevertheless, to simulate the behavior of a neural network, in the following, it is pretended that the top one third of the image always describes the background. To achieve this, the function takes besides

the image data (which is only relevant for an actual neural network) the distance between the user and the wind turbines. For this purpose, the GPS sensor of the device is addressed and the distances to the wind turbines are determined by a formula that also includes the curvature of earth. These distances are later used to fill an array with random values, the upper third with values greater than any of the distances after that the remaining space is divided by the number of wind turbines and for each wind turbine a equal part is filled with values that have a 50% chance of being less than the distance to that specific wind turbine. In this implementation the wind turbines have to be sorted from furthest away to closest to the user. Given this condition each wind turbine and the corresponding canvas element can be placed correctly.

To create the binary map described in Section 2.5.2, the previously created array of depth maps, each represented as a one-dimensional array, as well as the distance of the user to the wind turbines are passed to the following function. Within the calculation of the function *calcBinaryMap*, another array of one-dimensional arrays is now filled with values using a for-loop. This array of one-dimensional arrays is also defined as a global variable and the included arrays are initialized with the number of displayable pixels of the screen. As a result, the memory for the array does not have to be reinitialized in each run of the loop. Here, as described before, the value 1 is written to all locations of the pixels in the array that are in the foreground, 0 otherwise.

Finally, the function *applyBinaryMap* is executed, this takes both the array of image data from the object of the *trimHiddenCanvasData* function and the array representing the binary maps for each wind turbine. Again, this is done with the help of loops. Within a for-loop a while-loop is applied to each entry of the array. This while-loop checks each entry in the binary map. If the corresponding entry is unequal to 1, the pixel at the corresponding position in the array is set to transparent. Since each pixel is represented with four values (red, green, blue, alpha) and as described in Section 2.5.2, each pixel of the background in the overlay can be any pixel with 100% transparency, it is sufficient to set the alpha value to 0 at this point. Last but not least, the content of the overlay canvas element for each wind turbine is emptied and filled with the new corresponding image data. With this, the end of the *loop*function is reached. Afterwards, depending on the *videoRendering* variable, the loop is executed again or paused.

The code to support this described program flow can be found in Appendix B.

3.4. Performance Improvements

Runtime is (besides RAM) the most precious resource in this implementation. Since many loops are used in the implementation to get, manipulate and output the image data, it was important to keep the duration of each loop and function as short as possible.

For this purpose, the duration of individual functions shall be determined. Since the entire code of the augmentation is written in JavaScript, it was possible to fall back on proven methods for determining the duration of individual functions. For example, before calling a function, the value of $performance.now()$, which outputs the time

since accessing the page[12], can be assigned to a variable and then subtracted from the then-current value of $performance.now()$ after the function has been executed. An alternative to this would be to use $console.time(label)$ and $console.timeEnd(label)$. However, internally the same behavior is applied, yet it bears the problem that determining multiple time spans starting from the same point in execution requires one label each. By measuring the execution time, for example, the implementation of the $trimHiddenCanvasData()$ function was identified to be the longest-lasting and, ultimately, its duration could be reduced by more than 83%.

Another tool used to monitor the performance and internal progress of the app was Chrome Dev Tools which revealed that the included script for A-Frame loaded yet another file. To save the time and bandwidth it takes to load all the data, all the scripts were provided statically. This reduced the time it took to load the app, and also eliminated any internet access, so the app can now be used offline at 100%.

3.5. Results

In this part of the work, parts of the app are tested against different applications. For example, the usability on different display sizes is considered, as well as the simulation of multiple wind turbines and the calculation of the corresponding overlay canvas elements.

At the beginning, it should be mentioned that *one* system was used to simulate different display sizes for comparability reasons. Since mobile devices differ not only in their display size but also in the available computing capacity, the usage of different devices could distort the results.

Display Resolution[1]	Device Example	Total Pixels[1]	Time per loop (TPL)[2]	FPS[2]
360×640	BlackBerry Z30	$230,400$	26.728 ms	37
375×667	iPhone SE 1	$250,125$	28.580 ms	34
360×740	Samsung Galaxy S8+	$266,400$	30.336 ms	32
390×844	iPhone 12 Pro	$329,160$	38.311 ms	26
393×851	Pixel 5	$334,443$	38.256 ms	26
412×915	Samsung Galaxy S20 Ultra	$376,960$	43.568 ms	22
768×1024	iPad Mini	$786,432$	90.430 ms	11
820×1180	iPad Air	$967,600$	119.505 ms	8
912×1368	Surface Pro 7	$1,247,616$	137.739 ms	7
1024×1366	iPad Pro	$1,398,784$	155.171 ms	6

[1] measured in CSS Pixels; [2] implementation without neural network

Table 2. Comparison of frames per second based on different screen resolutions.

[12]https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

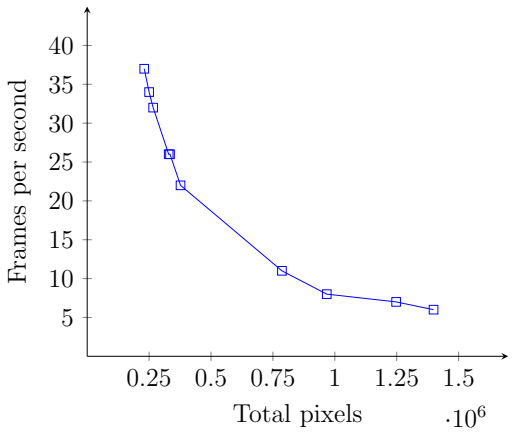

Figure 2. Relation number of pixels on device screen and achieved frames per second.

Please note that the average of 1000 iterations was used to smooth out fluctuations in performance.
The following table shows the average duration of a loop execution on a given resolution using different numbers of wind turbines.

	Number of wind turbines							
	2		3		4		5	
Resolution	TPL[2,3]	FPS[3]	TPL[2,3]	FPS[3]	TPL[2,3]	FPS[3]	TPL[2,3]	FPS[3]
360 × 640	38.190	26	48.234	20	63.061	15	69.602	14
375 × 667	41.056	24	55.372	18	65.878	15	77.561	12
360 × 740	44.144	22	28.751	17	70.478	14	88.137	11
390 × 844	55.655	17	77.601	12	89.951	11	100.141	9
393 × 851	58.940	16	73.218	13	86.577	11	103.614	9
412 × 915	67.179	14	81.978	12	104.149	9	116.896	8
768 × 1024	137.678	7	167.031	5	200.594	4	236.702	4
820 × 1180	173.815	5	202.708	4	249.754	4	290.003	3
912 × 1368	212.168	4	259.823	3	314.212	3	368.830	2
1024 × 1366	239.675	4	293.218	3	339.885	2	413.387	2

[1]measured in CSS Pixels; [2]in ms; [3]implementation without neural network

Table 3. Comparison of frames per second based on different amount of rendered wind turbines and screen resolution.

Here, it is to be considered that the duration of the calculation with a single wind turbine was not included in the table. The corresponding data can be found in Table 2. It should also be noted that the number of FPS has been floored. The following graph summarizes the results from Table 2 and 3 in terms of frames per second.

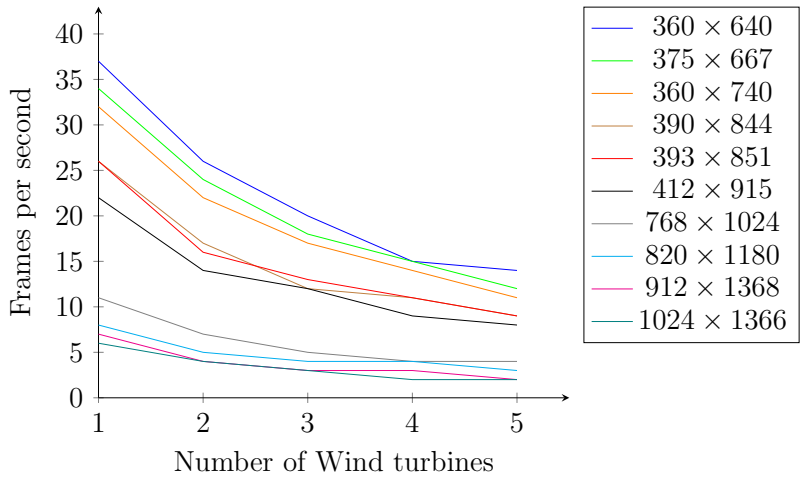

Figure 3. Relation between number of windturbines rendered and achieved frames per second based on screen resolution.

The four main functions of a loop described in Section 3.3 require different amounts of time. The following table shows an example of the duration in milliseconds as well as in percent of the total duration of these functions for a screen size of an iPhone 12 Pro.

#wt	1		2		3		4		5	
Func.	ms	%	ms	%	ms	%	ms	%	ms	%
tHCD	8.73	22.83	11.49	20.66	13.52	17.42	17.44	19.39	18.01	17.98
cDM	25.43	66.48	34,59	62.15	33,24	42.83	39.74	44.18	41.67	41.61
cBM	2.32	6.07	5,26	9.46	22.53	29.03	18.80	20.90	23.35	23.32
aBM	1.76	4.6	4,31	7.74	8.30	10.70	13.95	15.51	17.09	17.07
Total[1]	38,24	99.98	55,65	99.99	77,59	99.99	89.93	99.98	100.12	99.98

[1]Total times may not add up correctly due to rounding
#wt: Number of wind turbines, *tHCD*: *trimHiddenCanvasData*,
cDM: *calcDepthMap*, *cBM*:*calcBinaryMap*, *aBM*: *applyBinaryMap*

Table 4. Distribution of required time per function in loop

Again, 1000 iterations were considered to smooth out potential performance variations as much as possible. Furthermore, it should be noted that these values may not correspond to the corresponding total durations from Table 2 and Table 3. This is due to few additional auxiliary functions that must be executed during a loop.

In addition to looking at the frames per second achieved depending on screen size and the number of wind turbines to be rendered, the usability of a neural network that only calculates relative values was also tested.

To test the usability, a total of 20 videos were recorded, 10 of them with the help of a tripod, the others without. For the videos where no tripod was used, it was tried to keep the smartphone as still as possible. The duration of each video was about one

minute. They were recorded with the above-mentioned test device in a resolution of 1080×1920 pixels with 60 frames per second. The exact number of frames per video are documented in the table below:

Video	Videos with tripod		Videos without tripod	
	Duration in seconds	Number of frames	Duration in seconds	Number of frames
1	60	3571	60	3584
2	60	3598	60	3608
3	60	3580	60	3608
4	60	3635	60	3614
5	60	3558	60	3581
6	65	3832	60	3583
7	60	3576	60	3558
8	60	3573	60	3574
9	60	3587	60	3580
10	60	3577	60	3593
Total:	605	36,087	600	35,883

Table 5. Number of frames per video. Videos taken with and without a tripod.

Since the given durations are rounded, the number of frames may not result in sixtyfold the number of seconds.

Each individual image was then evaluated by the MiDaS neural network. Applying each model to the total 36,087 images of the videos with tripods took 1 hour and 40 minutes each. The evaluation of the images without tripod took about the same time. For comparability, all three available models of the network were used. Relative depths that the neural network could not determine are assigned a value of 0. It is important to know that there can also be negative values, this is because the network decides for each pixel its placement in relation to a previously selected pixel. The previously selected pixel is assigned a fixed value, but without knowing if there are more pixels that are relatively closer or farther away. Since the number of pixels for which no depth could be determined is important, the pixels assigned 0 as a value were counted. The network always gives the same result for the same image, yet the small differences between two individual images may have a great influence on the evaluation of the network. The most important key figures regarding 0-values per video and model can be found in Appendix D in Table 6 for the videos with tripod and in Table 7 for the videos without a tripod, respectively.

The calculation of the number of 0-values was as follows:

Let $|F|$ be the total number of frames, $|V|$ the total number of videos, and n the total number of pixels. For each frame $f \in F$ of each video $v \in V$ a list of relative depths is defined as $D_{vf} = [d_1, \ldots, d_n]$. Now a function $func$ can be defined which maps every value unequal to 0 to 1. The number of 0-values per frame is, therefore, defined by: $Z_{vf} = n - \sum_{i=1}^{n} func(D_{vf_i})$. The minimum per video is defined as $Z_v^{min} =$

$min\{Z_{vf} \mid f \in F\}$, and the maximum per video is defined as $Z_v^{max} = max\{Z_{vf} \mid f \in F\}$.

Another important indicator is the average value of the relative depth of each pixel and its dispersion. For this purpose, on the one hand, the average of the values of each pixel was determined, excluding the values that received the value 0 at least once (i.e., where no depth could be determined).
The average value of all pixels per video was calculated as follows:
Let $D'_v = [d'_1, \ldots, d'_n]$ where $d'_k = \prod_{i=1}^{|F|} D_{vi_k}$, with $k \in \{1, \ldots, n\}$ indicating the index of values in the list, be a list where every depth is index-wise multiplied across all frames. Consequently, every relative depth which at some point has taken the value 0 yields 0. Now a function can be defined which maps every value unequal to 0 to 1. Applying this function to the list D'_v creates a bit mask BM_v. Let $A_v = [a_1, \ldots, a_n]$ with $a_k = (\sum_{i=1}^{|F|} D_{vi_k})/|F|$, $k \in \{1, \ldots, n\}$. Index-wise multiplication of values was performed to discard the values which have been 0 in any frame, so that $A'_v = A_v \circ BM_v$. Later a function was defined which filters non-zero values from a list, leaving only the average values of pixels which never have been zero in any frame. These values were then summed up per video and its variance was plotted graphically in the form of box-plots in Figure 4, 5 and 6 for the videos with tripod and in Figure 7, 8 and 9 for videos without tripod in Appendix D. Here, only one graph was created per model for all videos. A direct comparison of the individual models against each other was not possible, since the calculated values of the relative depths were very different.

On the other hand, the graphs 10 to 69 in Appendix D recorded how much the pixel with the highest and lowest fluctuation changed in values. Here, only pixels were considered that had never been assigned a 0-value at any point in time. Including these values could bias the analysis due to inaccuracies in the neural network. Only the videos that were recorded with a tripod were used. Only these videos permit the assumption that differences in values do not result from a changed camera angle, position or orientation. The values of the relative depths were rounded to four decimal places.
Those values were calculated as follows:
Let $Var_v = [vard_1, \ldots, vard_n]$ with $vard_i$, $i \in \{1, \ldots, n\}$ being the variance over all relative depths at pixel i in video v. $Var'_v = Var_v \circ BM_v$ maps all variances of pixels that have been 0 at any point to 0. Var_v^{max} describes the pixel with the greatest fluctuation of depth values in video v where it is defined as follows:

$$Var_v^{max} = max\{vard_i \mid i \in \{1, \ldots, n\},\ vard_i \in Var'_v\}.$$

Var_v^{min} describes the pixel with the least fluctuation of depth values in video v where it is defined as follows:

$$Var_v^{min} = min\{vard_i \mid i \in \{1, \ldots, n\},\ vard_i \neq 0,\ vard_i \in Var'_v\}.$$

Var_v^{max} and Var_v^{min}, therefore, indicate not only the value of the most and least fluctuating pixel but also the corresponding indices. To plot the values of all frames at said indices, they were added to the corresponding lists VAR_v^{max} and VAR_v^{min}.

The graphs of the values of the most and least fluctuating pixel per video were not plotted together, neither were the values of different models, because the scales of the Y-axis (the values of the relative depths) are in some cases very different.

These investigations thus conclude the test part of the work.

3.6. Discussion of Results

In this part of the paper, the results of the tests described in the previous part will be discussed. From Table 2 it can be seen that the number of achievable images per second depends on the screen size used. This results from the fact that the calculations of the determined distances or the determination of whether a pixel is in the foreground or not, must be executed more frequently. It also follows from Table 2 that there is also a correlation between the number of achieved frames per second and the number of rendered wind turbines. The more wind turbines are used, the lower the number of frames per second. This follows from the same argument as for screen size. When multiple wind turbines are used, it follows in the implementation that multiple foregrounds must be computed per iteration. Even though the depth of the pixels has to be determined only once when using X wind turbines, it has to be determined X times whether the pixels are in front or behind the respective wind turbine. Likewise, it must be determined X times whether the pixel should thus be visible to the user or not.

The results of testing several wind turbines from Table 3 strongly suggest that no more than two wind turbines should be loaded or displayed at the same time. When using more than two wind turbines, the number of frames per second achieved drops drastically and is no longer usable without a tripod.

However, this finding should be taken with a grain of salt. The use of a suitable neural network, which can determine absolute distances per pixel, could influence the time per loop both positively and negatively. If only one wind turbine is to be displayed, the duration of the computation of simulated depths has the greatest impact on the total duration. If multiple wind turbines are to be displayed, the duration of foreground computation will predominate. When using five wind turbines, this takes up to 70% of the total duration (with the resolution of an iPad Pro). Thus, an efficient neural network could only improve the archived frames per second for a very limited number of wind turbines. For several, an efficient neural network would have less impact on the required time per loop.

It could also be considered whether the wind turbines should be dynamically loaded into the app and dynamically deleted again. This would ensure that only the wind turbines within a certain radius around the viewer are loaded. Through this dynamic, the user can then walk through the virtual wind farm and always view the closest wind turbines on his mobile device. This dynamic can be achieved by setting the ”maxDistance” attribute of the *a-camera*. In the current implementation, a value of 10,000 meters is used.

Even though no suitable neural network has been found that can calculate the absolute depths of each pixel, the use of the MiDaS network has been investigated in more

detail. The preliminary tests to check the general usability have already shown that the dispersion of the values of the relative depths from image to image is considerable. This is also evident from the sometimes strongly varying number of 0-values seen in Table 6. The more accurate the network, the more 0 values are output. This is because even smaller areas are examined more precisely. The potential use of the relative distances to place wind turbines is complicated by the appearance of 0 values. Since any value less than the distance value of the wind turbine is interpreted as foreground, the sometimes large variation in the number of 0 values makes it impossible to accurately represent the foreground. Since all videos from Table 6 were taken with a tripod, only small fluctuations were expected here. Table 7 shows the number of 0 values for the videos taken without a tripod. Consequently, it is noticeable that the differences between the minimum and maximum of the 0-values are considerably larger than in the videos with a tripod. Also, the span, visible in the next-to-last column, is significantly higher. Surprising is the evaluation from the video 9 with the MiDaS model *small*, here there was at least one frame where the network could assign a value to all pixels, thus there was no 0-value, however, the average of the 0-values of the frames of the video is more than 30% of all pixels.

Apart from the number of indeterminable depths, the box-plots of Figure 4, 5, and 6 also show that the values of the non 0 values vary greatly. In the boxplots mentioned above, the range of values per video can be seen, this gives information about how close or far apart the values within a video are from each other. From the box-plots of Figure 7, 8 and 9 it can be seen that the range increases for the videos without tripod. Also, the number of outliers is much higher.

A look at the Graphs 10 to 69 gives a detailed insight into the fluctuations per video regarding the strongest and weakest fluctuating pixel per model. However, it is noticeable that the values for the least fluctuating pixel are not meaningful, as they are close to 0 in almost all videos and models. The strongest fluctuations per video and model, however, give a good insight into how sensitive the models are concerning marginal changes in the camera image between two consecutive frames.

If the values did not differ so much it would have been conceivable to take a snapshot, using the cell phone on a tripod, simulating the distance of the wind turbine by the relative depth at the expected location in the image. Then the app could have been rebuilt and transferred back to the phone. An implementation by touching the screen would have been conceivable, but the substantial differences between the relative depths of each pixel from frame to frame suggest that even touching the screen would already affect the calculation of future depths in the video. Due to the time required to determine the depth of an image, the need for an Nvidia graphics card to run the network (at least the more accurate models, *hybrid* and *large*) on the computer, and the fact that the values of the same pixels can sometimes change significantly between frames, this approach is not useful and thus not pursued. Nevertheless, it can be said that a filter for the values describing an invalid or non-calculable distance would be useful. Since this value can be different between networks, this value must be adapted respectively. The use of a neural network that only calculates relative values is therefore not useful.

Even though these results suggest that a neural network that determines only relative values is not suitable, requirements and criteria for a suitable network can be determined. Through the TPL in Table 2 and 3 the desired maximum inference time of a network can be determined. Depending on the desired display size and the resolution in CSS pixels, respectively, it can be said that when simulating only one wind turbine on an iPhone 12 Pro, the inference may take a maximum of 28.766 ms. This value results from the fact that the video should ideally still be displayed with 24 frames per second, therefore the TPL may take a maximum of 1000ms/24 = 41.667 ms, yet the current implementation already requires 38.311 ms, thus leaving a maximum of 3.336 ms for the inference of the network. However, the duration of the generation of random values must be subtracted (25.43 ms see Table 4), resulting in a total of 3.336 ms + 25.43 ms = 28.766 ms for the inference. Using two wind turbines, inference should already take only 20.607 ms; using more than two wind turbines, according to Table 4, no suitable network would be able to achieve the desired 24 frames per second on any screen size.

This concludes the discussion of results.

3.7. Testing

In this part of the paper, new findings that emerged during the code writing process are discussed. In addition, problems with the implementation of the previously desired features are also mentioned and described here.

3.7.1. Insights

Previous versions of AR.js repositioned the virtual objects every time the GPS data was updated. Since the rotation and GPS sensors do not always provide accurate data, this behavior caused the virtual elements to frequently change their position in the image. This could be worked around by only periodically requesting the user's GPS data and then using the "simulateLatitude" and "simulateLongitude" attributes of the *a-camera*, manually updating the user's position. Since version 3.3.1, this problem can be overcome more comfortably, the property "smoothingFactor" exists in the "arjs-look-controls" component, which can be added to the *a-camera*-tag [45]. The value of this property specifies the percentage for the exponential smoothing with which a new read value of the sensors influences the rendering. If k is the passed value of the property, the last calculated value of the sensors has an influence of $1 - k$ with $k \in [0, 1]$. The exponentially refined display angle of the virtual object is thus:

$$\text{smoothedAngle} = k \cdot \text{newValue} + (1 - k) \cdot \text{previousSmoothedAngle}.$$

Another important insight was that a website's unit of measurement for pixels on a screen does not have to match the actual physical pixels of the device. The so-called CSS pixels do not exactly correspond to one physical pixel per unit, because they try to display content in the same way on different devices with high-density displays using the new property "pixel-ratio". By using this unit of measurement, it could also

happen that the video provided by AR.js or the device's camera is too big for the screen. Should this be the case, it had to be guaranteed that the middle part of the image is always visible. Shifting it to the left or right would irritate the user. Even though AR.js handles this task on its own, it is something to consider when inserting the overlay-canvas element into the DOM. A canvas element has the general behavior, that even if the content of the element is larger than the actual width of the element, this content is compressed to fit inside the element.

3.7.2. Identified Challenges

Challenges arose in all areas of implementation, starting with the choice of an appropriate version of AR.js. The last official stable version was released in May 2021, after which a new beta version was released in November 2021, and currently (as of March 9, 2022) version 3.4.0 is available as an alpha version. AR.js is maintained by the "AR.js organization". There are eight people in the GitHub organization, but only three of them are actively working on the repository. With over 151 open issues (as of March 9, 2022), the current version is not expected to work flawlessly.[13][14]

As best performance is always desirable, the latest stable version (3.3.3) was used in the implementation. However, it should be noticed that this version is not compatible with the latest version of Aframe (1.3.0 as of March 3, 2022)[15]. Nevertheless, since only basic functions of Aframe were needed in the implementation, it was decided to use version 1.0.4 from February 2020, being the best working in this case.

Facing possibly poor internet connection in rural areas of Germany, the required data size was investigated. The minimized version of Aframe 1.3.0 is just over a megabyte in size. Itself needs one more file to work, namely the "aframe.min.js.map". This is just under three megabytes in size. The AR.js file used is confusingly named "aframe-ar-nft.js" and is itself also more than two megabytes in size. In total, the data needed to successfully render the wind turbine model is more than six megabytes in size. To counteract possible poor internet connections in the countryside, the files were provided statically as part of the app, resulting in no need to be downloaded every time the app is started. The only disadvantage that this approach could bring on, would be that the latest available version of the individual components is not automatically used. When a new stable version is released, it should be considered updating the app.

Another challenge was the partly inconsistent performance of the rendering of wind turbines. Depending on the computing power of the mobile device, the calculation of the foreground may take up a large part of the device's computing resources. This can cause the animation of the spinning wind turbine to temporally freeze. Another aspect that can occur is that the rendering of the wind turbine may take longer because the computing capacity is occupied by the calculation of the foreground. To address these two problems, two buttons have been integrated into the app. One button switches

[13] https://github.com/kalwalt
[14] https://github.com/AR-js-org/AR.js/
[15] https://github.com/AR-js-org/AR.js/issues/385

off the calculation of the foreground, or, if it is already switched off, activates it again. The other button determines whether the rendered wind turbines rotate or not.

Yet, the solution of the previous problem created new challenges. It was assumed that since the wind turbine is created via a component, one could read and manipulate the attributes of these components defined in the schema. Again, to separate the logic from the actual operation here, the HomePage triggers an event in the iFrame. However, reading the values via the function "getAttribute('windturbine')" (this was the name of the former model for creating a wind turbine), turned out to be more difficult than expected. By definition, the function returned a string, but at runtime, this was an object. By concatenating "JSON.parse(JSON.stringify(attribute))" the values like "speed", "heightScale", "rotorScale", "direction" as well as "rotation" of type Boolean (used at that time to indicate if the wind turbine is rotating or not) could be read. After manipulating the value of "rotation", there was, unfortunately, no way to return these values to the component of AFrame. So, finally, this way was discarded.

To overcome this challenge, a new wind turbine model was created that is an exact copy of the first model, but without the rotation function and its execution at each *tick*. This is managed in a way that internally the status is handled whether currently, the wind turbine should rotate or not. Depending on this status, the appropriate model is loaded. If the status changes, all occurrences of wind turbines are deleted, and the now correct model is loaded. Since the models are not large and AFrame is well equipped for this task, this is a very performant solution. However, the only disadvantage this approach has, is that the blades of the wind turbine do not hold in the current position, but are always drawn in a position defined in the model.

A new challenge arose taking different screen sizes into account. Using landscape mode on a mobile device only makes sense if the width of the screen is still large enough to display the content adequately. This is only the case on a tablet, but not on a cell phone. Secondly, it was identified that there are problems with the correct functionality of AR.js[16] when using the landscape mode. For these reasons, the use of landscape mode on the Android operating system was not granted. This was achieved by adding an 'android:screenOrientation="portrait"' entry to the AndroidManifest.xml file.

Addressing the performance and power consumption of the device the number of sensor request has been investigated. While the exact position of a virtual object is given by the specification of longitude and latitude, the current position of the user can be determined by the GPS sensor in the device. Thus, the direct approach would be to re-determine the user's position and recalculate the distance to the virtual object every time the position is updated, or after a certain time interval. However, in AR.js, the custom attributes "distance" as well as "distanceMsg", have been implemented for all objects with a *gps-entity-place* attribute. Since AR.js requests the user's position anyway, using these attributes would be more resource-efficient than separately making a request to the GPS sensor and performing the calculation manually. Unfortunately, using these attributes did not work and no explanatory description was provided by the

[16]https://github.com/AR-js-org/AR.js/issues/376

developers, so it was decided to use the manual approach. However, this is a possible improvement, potentially achieved by using a newer version of AR.js.

Within the described implementation in Section 3, the *loop* function is called constantly. However, tests have shown that this can be very resource intensive. A short delay before the next start of the function solves the problem here. This is at the expense of the possible maximum number of frames per second, but an increase in FPS is achieved compared to the constant execution of the function. What seems contradictory at first glance can be justified by the fact that JavaScript is not inactive during the delay time but cleans up the memory of variables that are no longer used. While testing, a loop delay of just five milliseconds has proven to be sufficient here.

Finally, an improvement for user feedback was added to the implementation. The initialization of AR.js and the preparing functions takes about 2.59 seconds. To visualize the current state of the app, a loading spinner as well as the current process of the app are displayed.

3.7.3. Identified Problems

In this part of the work challenges are mentioned that have not been able to be overcome during implementation, and therefore being considered problems.

In Section 2.2.2, the feature of being able to see where the closest wind turbine is positioned, with the help of markers on a map, was mentioned. While researching how this feature could have been implemented, it was noticed that AR.js does not give the user or the programmer any information about where the virtual object is located on the screen. Since the virtual object is like a layer on top of the video, these two elements are unrelated to each other. The attempt to determine the position of the wind turbine, in the same way as the position of the video on the screen, cannot be successful because the wind turbine element in the HTML-code occupies a size of 0×0 pixels on the screen, in other words, it should not be visible. However, since the wind turbine can still be seen when the user holds the phone in the right direction, it is due to the internal implementation of AR.js and its use of the *a-scene*. It is not intended to completely rule out the possibility of this feature, especially since there might be a way with the underlying library AFrame or even Three.js. However, no such possibility was found during the conducted research.

The next problem affected another desirable feature of the app. The screenshot feature was intended to allow users to take a snapshot of the image from their mobile device to save the image of the rendered wind turbine and show it to friends and family if necessary. For this purpose, three different approaches were pursued. A library called ”html2canvas”[17], as well as a plugin from the official Capacitor community called ”Screenshot”[18] and the possibility to capture the current image of the screen with the ”Screen Capture API”[19] of the browser.

[17] https://html2canvas.hertzen.com/
[18] https://ionicframework.com/docs/v3/native/screenshot/
[19] https://developer.mozilla.org/en-US/docs/Web/API/Screen_Capture_API/Using_Screen_Capture

html2canvas is a TypeScript library developed and maintained by Niklas von Hertzen with more than 25 thousand stars on Github. The script can either be installed with the Node Package Manager or inserted as an external file via a script tag. To use the script simply search for the element to be saved and pass it to the function of the same name as an argument. A canvas object is created, which can then be either displayed on the screen or saved to the gallery. The problem that occurs when it is called in the HomePage.vue file is that the actual video is not in the body of this file, but in the iFrame that is included in the HomePage. Thus, calling the function on the body element of the HomePage will only output the buttons, as well as the header on a black background. When trying to call the function in the file included by the iFrame, different problem occurs. The canvas element used by AR.js to display the wind turbine and the entire video has a "webgl" context. This context does not allow taking a snapshot of the current state of the element as long as the "preserveVideoBuffer" attribute is set to *false*. Unfortunately, this attribute cannot be manually changed for the moment of the screenshot and then reset to the initial value. AR.js uses this option to improve the performance of working with virtual objects. The only content visible in the screenshot is the OverlayCanvas, as this has a "2d" context. However, another problem arises here, since the video can take on very large dimensions, the screenshot also has such large dimensions with the non-displayable area being filled solid white. For all these reasons, the library of Niklas von Hertzen cannot be used.

The official community plugin from Capacitor called "Screenshot" looks like a simple solution at first sight. Capacitor is the underlying plugin manager of Ionic. The plugin has to be installed via two commands in the terminal, then the type "Screenshot" has to be imported and instantiated, finally, the function *save()* has to be called on the Screenshot instance. Here the function takes three further parameters to determine the format, quality, and name of the screenshot. However, two things quickly stood out that diminished the hope of success. Firstly, there is no documentation of the plugin being integrated and used in a Vue framework. Secondly, this plugin has not been further developed for over five years and was finally archived. After installing the plugin and importing and instantiating it the same way as, for example, the "html2canvas" library, a problem when saving an image occurred, which is also noted on the project's issue page on Github. The content of the iFrame is solid black. Only the buttons and the header are visible on the image. The implementation can be found in Appendix C. Since no solution to this problem was proposed on the GitHub page, this plugin is also not a solution for implementing a screenshot function.

Another option was to use the browser's native Screen Capture API. At the latest since the rapid increase in the use of the virtual meeting software "Zoom" and the possibility to share one's screen, the usage of this API has become widespread. It allows the user to share either their entire screen or just a single window on the web. The idea of how this API can be used to take a screenshot was the following. When the user presses the screenshot button, the API is called, which in turn shares the screen (with the app itself). Then, using the ImageCapture interface [20], the current image of

[20]https://developer.mozilla.org/en-US/docs/Web/API/ImageCapture

the shared screen can be captured and saved to a canvas element, for example. To use the interface the W3C Image Capture definitions[21] had to be installed via npm and then manually added to the *tsconfig.json* file under "compilerOptions → types". On the computer, the problem arises that a prompt is started asking which window should be shared, and only then a screenshot is saved. Even if this behavior works almost as desired on the computer, other difficulties arise on the smartphone. Here, screen sharing is not supported via this API. Whether this will change in the future is still uncertain. The implementation of this idea can be found in Appendix C.

The last idea was to find another way to use JavaScript to trigger the native screenshot functions of mobile devices. This would mean that the status bars at the top of the screen, showing the network provider, notifications and the battery, and possibly the virtual navigation buttons at the bottom would be visible on Android phones, but it was decided to put up with this behavior to be able to take screenshots. Unfortunately, research regarding such a function or library was inconclusive.

Due to the multitude and diversity of problems encountered regarding the screenshot function it was not possible to implement it, although this feature can be very important for the user. However, the user still has the option to take a screenshot of the app via the phone's native key combination.

Finally, the last problem affects the representation and correct order of wind turbines to the user. Since for performance reasons the overlay-canvas elements are created only once and afterward their content is only rewritten, the order of these elements in the DOM is fixed. To identify which canvas element belongs to which virtual wind turbine, identifiers were used to clarify an affiliation. The IDs of the wind turbines are also not modifiable. They are already defined in the windturbine.json file. From these two aspects, the problem arises that the order of the wind turbines is not modifiable. Since the wind turbines should be displayed in descending distance to the user, this is not a problem as long as the user is standing in line with the wind turbines. If the user now moves and the order given by the distances to the wind turbines changes, the current implementation fails to display the wind turbines correctly. Even if this problem is known, the solution does not seem to be so directly possible. An attempt to manage the wind turbines internally (after a one-time load from the JSON file) failed because of moving single HTML elements in the DOM while persisting their state was not possible. The intent to delete and reload them was also unsuccessful. There is no known library that would solve this specific problem. Since the manual attempt was without success, this problem was not treated further.

[21] https://www.npmjs.com/package/@types/w3c-image-capture

4. Conclusion

During this Bachelor thesis, it became apparent that the model pictured in Section 2.2.1 was a very direct attempt to represent the course of the app and the interaction between the work on the computer as well as on the mobile device. The newly gained insights, through implementation as well as testing, were course changing. The lack of a neural network for estimating absolute distances also eradicated almost entirely the work on the computer for the course of this Bachelor's thesis.

New findings during the implementation of the app in JavaScript pointed out the peculiarities of the popular augmented reality framework AR.js. Nevertheless, these could be handled successfully and the advantages and strengths of AR.js could be used. Thus, the final model mainly refers to the flow of the functions from Section 3.3. Components such as the segmentator and blender have been custom written to fit their special task.

Reviewing the success of the app, it can be checked against the previously desired properties from Section 2.2. Regarding the criteria from Section 2.2.1, it can be said that all essential criteria were met. The app works on the Android operating system and no libraries or plugins were used that would stand in the way of using it on iOS. AR.js takes care of the task of correct positioning, displaying only currently visible virtual objects, as well as checking the GPS position of the user. Furthermore, it was described in Section 3.7.1 that by using local copies of the required libraries, the app works completely offline. It should also be noted that only the JavaScript programming language was used. Only the determination of the pixels that are in the foreground did not initially work as expected. Nevertheless, a black box implementation was created for this, into which the integration of a suitable network should be simple.

Regarding the features described in Section 2.2.2, it can be said that some could already be checked off by using the AR.js framework. For example, the reading of the sensors is handled by the library. Looking at the mentioned frames per second from Table 2 and 3, it can be claimed that the performance of the app is as expected. Regarding the non-essential features, not all of them were successfully implemented. For example, no arrows are to be found in this implementation to indicate the next wind turbine, nor was a screenshot function implemented.

All in all, it can be said that the essential parts of the work have been incorporated into the final model.

4.1. Identified Requirements for Neural Network

Even though, as mentioned before, no neural network was found for the accomplishment of the task within the scope of this work, characteristics can be established, which a suitable neural network should fulfill, as well as possibilities to reduce its workload.

The methods for improving the performance from Section 3.4 as well as the results of the test from Section 3.5 have also shown, among other things, which time limitations exist for the neural network. One run of the loop described in Section 3.3 takes 38.311 milliseconds on a screen size of an iPhone 12 Pro (as an average of 1000 executions).

Given this value, it corresponds to roughly 26 frames per second. It should be noted that this value may depend on the end-device and other factors, such as the number of simultaneously opened apps.

To be able to present a smooth image to the user, i.e., at least 24 frames per second, the inference of the neural network should therefore take a maximum of 28.766 milliseconds, as calculated in Section 3.6. It should be noted here that faster is always better. However, often (yet not always) a trade-off of accuracy and speed of inference is noticed in neural networks [46]. Since accuracy is also important, a suitable implementation must be found here as well.

The neural network should also be able to handle image data, respectively their representation in the form of a one-dimensional array. Furthermore, in the implementation from Section 3.3, the output of the network is also calculated in the form of a one-dimensional array. Any other numerical representation of depth is also conceivable here. The original format of the neural network, as well as the framework in which it was written, are almost irrelevant here, since, as mentioned in Section 2.4.3, there are sufficient converters.

Possible improvements of a neural network, regarding the performance, would be a reduction of the pixels to be evaluated. Here, for example, four pixels could be combined into one. Four pixels are to be chosen here, which together form a square. Each value of the RGBA channels could form the new pixel in equal proportions. Mathematically expressed this would mean:

Let $P^* = (R^*, G^*, B^*, A^*)$ be the new pixel to be created with $R^*, G^*, B^*, A^* \in \{0, \ldots, 255\}$ from pixels $P_0,\ P_1,\ P_2,\ P_3$ then $R^* = \frac{1}{4}\sum_{i=0}^{3} R_i$. G^*, B^* and A^* respectively.

Having then determined the depth for that combined pixel, it must be applied to all four original pixels. This generalization results in a certain inaccuracy but at the same time, a reduction of the depths to be calculated by 75% is achieved. An alternative would be to combine only two neighboring pixels, so the reduction would still be 50%, but the inaccuracy would also be reduced by 50%.

Another feasible improvement of the neural network would be to apply it only to the part of the image on the screen where the virtual object is being rendered. Even though this would drastically improve the performance of the network, since the network would not have to be applied at all when the virtual object is not visible, no possible way was found during testing to read the position of the virtual object in the video, see Section 3.7.3.

A simpler, more realistic improvement in neural network performance would be to not apply the network to every frame. With up to 26 frames per second as described earlier, applying it to every second or even fourth frame would be conceivable. By suspending the application of the neural network, the app saves the resources of some inferences. Thus, the duration that the neural network is allowed to take for inference is increased, accordingly. Alternatively, but with the same effect, it could be imaginable not to apply the network again if the cell phone has not moved much. However, the problem here is that close objects move more than distant ones relative to the user when the position of the camera changes. This could result in the depth determination

being inaccurate and would also not be updated as long as the change of position does not surpass a chosen threshold.

Lastly, it should be noted that the model of the network should only be loaded once. After that, it occupies memory in the RAM of the mobile device. For this reason, the model should not be too large. For example, the most accurate MiDaS network is over a gigabyte in size and thus unsuitable for use on a mobile device. Even though the size of RAM on mobile devices has increased over the past few years, it doesn't mean that an app will have access to the entire memory at any time. It must also be considered that the size of the app grows due to the integration of the model. Thus, the memory must be sufficient not only in RAM but also in the internal storage of the device.

4.2. Future Work

Probably the most important part for future work on this project is the creation of a suitable network. To achieve this, images of suitable scenery must be taken along with the absolute depths of each pixel. Furthermore, other features discussed but not implemented can be addressed. Once this is accomplished, possible applications of the app are far-reaching. Since no object recognition is used, other construction projects such as larger buildings, bridges, or roads can also be augmented without much further ado. However, the application here is not limited to the construction aspect. Computer-animated 3D movies, as well as playful applications, could also be made more interesting by applying the described logic.

In summary, this thesis pointed out the importance of a visual representation tool for planned construction projects for social acceptance and provided a detailed description and implementation of one possible solution.

Different approaches were carefully evaluated, and the decision was made to use a neural network. Thoughtfully considering different networks, it turned out in the course of the work that such a network does not (yet) exist. Nevertheless, in order to do the preliminary work for the later integration of such a network, a black box for the use of a neural network for absolute depth estimation was implemented. At the same time, the challenges and problems during the implementation were addressed and documented. By simulating the output of a possible network, the feasibility of the project was successfully proven.

Overall, through the findings of this work, it can be said that augmenting reality to visualize wind farms including obstacle detection is conceivable.

References

[1] UCSUSA. The Hidden Costs of Fossil Fuels, 2016.

[2] Fang Dong and Li Hong. Clean development mechanism and nuclear energy in china. *Progress in Nuclear Energy*, 37(1):107–111, 2000. ISSN 0149-1970. doi: 10.1016/S0149-1970(00)00033-0. Global Environment and Nuclear Energy Systems-3 Proceedings of the Third International Symposium.

[3] Akhilesh Yadav, Ajeet Singh, and A. Shukla. *Nuclear Energy and Conventional Clean Fuel*, pages 23–44. Springer Singapore, Singapore, 2022. ISBN 978-981-16-4505-1. doi: 10.1007/978-981-16-4505-1_2.

[4] Martin J. Goodfellow, Hugo R. Williams, and Adisa Azapagic. Nuclear renaissance, public perception and design criteria: An exploratory review. *Energy Policy*, 39(10):6199–6210, 2011. ISSN 0301-4215. doi: 10.1016/j.enpol.2011.06.068. Sustainability of biofuels.

[5] Andreas Burger, Benjamin Lünenbürger, and Dirk Osiek. Sustainable electricity for the future, 2012.

[6] Bundesministerium für Wirtschaft und Energie. Erneuerbare Energien, 2021.

[7] Umwelt und Verbraucherschutz Nordrhein-Westfalen Landesamt für Natur. Energieatlas NRW, 2020.

[8] Bundesministerium für Wirtschaft und Energie. Das Erneuerbare-Energien-Gesetz, 2021.

[9] United Nations Climate Change. COP26 KEEPS 1.5C ALIVE AND FINALISES PARIS AGREEMENT, 2021.

[10] European Commission. REPowerEU: Joint European action for more affordable, secure and sustainable energy, 2022.

[11] Maarten Wolsink. *Wind power: basic challenge concerning social acceptance*, pages 1785–1821. Springer, New York, NY, 09 2013. ISBN 9780387894690. doi: 10.1007/978-1-4614-5820-3_88.

[12] Erwin Hofman. Social Acceptance of Renewable Energy, 2015.

[13] Benjamin Wehrmann. Limits to growth: Resistance against wind power in germany, 2019.

[14] Johannes Pohl, Franz Faul, and Rainer Mausfeld. Belästigung durch periodischen Schattenwurf von Windenergieanlagen, 1999.

[15] Eivind Skaaland and Kelly Pitera. Investigating the use of visualization to improve public participation in infrastructure projects: how are digital approaches used and what value do they bring? *Urban, Planning and Transport Research*, 0(0): 1–15, 2021. doi: 10.1080/21650020.2021.1887757.

[16] Christine Barton, Lara Koslow, and Christine Beauchamp. How Millennials Are Changing the Face of Marketing Forever, 2014.

[17] Michael Schornstheimer. Wenn Ingenieure philosophieren, 2016.

[18] Ellen DaSilva. Niantic Labs and Pokemon Go: Bringing Ar to the Masses, 2017.

[19] 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. URL `https://doi.org/10.1088/1742-6596/749/1/012020`.

[20] Energie Baden-Württemberg AG. EnBW entwickelt Augmented Reality-App für die Visualisierung von Windkraftanlagen, 2021.

[21] Google. Motion tracking, 2021.

[22] Google. ARCore supported devices, 2021.

[23] Geobasis NRW Land Nordrhein Westfalen, Bezirksregierung Koeln. Digitales Geländemodell - Gitterweite 1 m - Paketierung: Einzelkacheln, 2022.

[24] Mozilla. WebXR Viewer, 2018.

[25] Igor Vasiljevic, Nick Kolkin, Shanyi Zhang, Ruotian Luo, Haochen Wang, Falcon Z. Dai, Andrea F. Daniele, Mohammadreza Mostajabi, Steven Basart, Matthew R. Walter, and Gregory Shakhnarovich. DIODE: A Dense Indoor and Outdoor DEpth Dataset. *CoRR*, abs/1908.00463, 2019. URL `http://arxiv.org/abs/1908.00463`.

[26] Jonas Uhrig, Nick Schneider, Lukas Schneider, Uwe Franke, Thomas Brox, and Andreas Geiger. Sparsity Invariant CNNs. In *International Conference on 3D Vision (3DV)*, 2017.

[27] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The Cityscapes Dataset for Semantic Urban Scene Understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[28] Eda Kavlakoglu. AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?, 2020.

[29] Huan Fu, Mingming Gong, Chaohui Wang, Kayhan Batmanghelich, and Dacheng Tao. Deep Ordinal Regression Network for Monocular Depth Estimation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[30] Jin Han Lee, Myung-Kyu Han, Dong Wook Ko, and Il Hong Suh. From big to small: Multi-scale local planar guidance for monocular depth estimation. *arXiv preprint arXiv:1907.10326*, 2019.

[31] Armin Masoumian, Hatem A. Rashwan, Saddam Abdulwahab, Julian Cristiano, and Domenec Puig. Gcndepth: Self-supervised monocular depth estimation based on graph convolutional network, 2021.

[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[33] Pablo Díaz. Pytorch is growing, Tensorflow is not., 2020.

[34] Saqib Shah. Microsoft and Facebook's open AI ecosystem gains more support, 2017.

[35] Microsoft. Microsoft onnxjs repository, 2020.

[36] Microsoft. Convert TensorFlow model to ONNX, 2021.

[37] Lu-Hsuan Chen. How to Convert Your Keras Model to ONNX, 2019.

[38] Eliot Andres. Converting a Caffe model to TensorFlow, 2017.

[39] argon.js Organization. argon.js GitHub Repository, 2017.

[40] argon.js Organization. argon.js Documentation, 2017.

[41] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2020.

[42] Armin Masoumian, David G. F. Marei, Saddam Abdulwahab, Julián Cristiano, Domenec Puig, and Hatem A. Rashwan. Absolute distance prediction based on deep learning object detection and monocular depth estimation models. *CoRR*, abs/2111.01715, 2021. URL `https://arxiv.org/abs/2111.01715`.

[43] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL `http://arxiv.org/abs/1405.0312`.

[44] Khronos. glTF RUNTIME 3D ASSET DELIVERY, 2022.

[45] Jerome Etienne. AR.js Documentation, 2022.

[46] Rohith Gandhi. Improving the Performance of a Neural Network, 2018.

A. AR.js Setup

```
<script src="https://raw.githack.com/AR-js-org/AR.js/master/aframe/
    build/aframe-ar-nft.js">
<script src="https://raw.githack.com/AR-js-org/AR.js/master/three.js/
    build/ar-nft.js">
```

Listing 1. Necessary script imports for AR.js

```
<body style="margin: 0; overflow: hidden">
  <a-scene
  vr-mode-ui="enabled: false"
  arjs="sourceType: webcam; debugUIEnabled: false;"
  >
    <a-camera
    gps-camera="gpsTimeInterval:1500"
    minDistance="20"
    maxDistance="10000"
    rotation-reader
    arjs-look-controls="smoothingFactor: 0.1"
    >
    </a-camera>
  </a-scene>
</body>
```

Listing 2. Settings for a-camera

```
AFRAME.registerComponent("windturbine-rotating", {
  schema: {
    speed: { type: "number", default: 0.02 },
    heightScale: { type: "number", default: 1 },
    rotorScale: { type: "number", default: 1 },
    direction: { type: "string", default: "south" },
  },
  init() {
    this.el.addEventListener("model-loaded", () => {
      const windTurbineMesh = this.el.getObject3D("mesh");
      this.positionMesh(windTurbineMesh);
      const towerMesh = windTurbineMesh.children.find(
      (element) => element.name === "tower"
      );
      this.setupTower(towerMesh);
      windTurbineMesh.children.forEach((element) => {
        this.setupRotorHeight(element);
        this.setupBlades(element);
      });
      this.el.isWindModelLoaded = true;
    });
  },
  setupBlades(element) {
    const bladeRegExp = new RegExp("blade.*$");
    if (bladeRegExp.test(element.name)) {
      element.scale.x = this.data.rotorScale;
```

```
      element.position.y += 2;
      this.data._windTurbineBlades
      ? this.data._windTurbineBlades.push(element)
      : (this.data._windTurbineBlades = [element]);
    }
  },
  setupTower(element) {
    element.scale.z = this.data.heightScale;
    const size = new window.THREE.Box3().setFromObject(element).
       getSize();
    this.el._windTurbineRotorYPosition = size.y;
  },
  setupRotorHeight(element) {
    if (element.name !== "tower") {
      element.position.y = this.el._windTurbineRotorYPosition;
    }
  },
  positionMesh(windTurbine) {
    switch (this.data.direction) {
      case "south":
      return windTurbine.rotation.set(0, 0, 0);
      case "north":
      return windTurbine.rotation.set(0, (180 * Math.PI) / 180, 0);
      case "east":
      return windTurbine.rotation.set(0, (90 * Math.PI) / 180, 0);
      case "west":
      return windTurbine.rotation.set(0, (-90 * Math.PI) / 180, 0);
      case "northwest":
      return windTurbine.rotation.set(0, (-135 * Math.PI) / 180, 0);
      case "northeast":
      return windTurbine.rotation.set(0, (135 * Math.PI) / 180, 0);
      case "southeast":
      return windTurbine.rotation.set(0, (45 * Math.PI) / 180, 0);
      case "southwest":
      return windTurbine.rotation.set(0, (-45 * Math.PI) / 180, 0);
    }
  },
  rotateBlades() {
    if (!this.el.isWindModelLoaded || !this.data._windTurbineBlades)
       return;
    this.data._windTurbineBlades.forEach((blade) => {
      blade.rotation.z -= this.data.speed;
    });
  },
  tick() {
    this.rotateBlades();
  },
});
```

Listing 3. Schema and setup to register a rotating wind turbine component

```
AFRAME.registerComponent("windturbine-still", {
  schema: {
```

```
    speed: { type: "number", default: 0.02 },
    heightScale: { type: "number", default: 1 },
    rotorScale: { type: "number", default: 1 },
    direction: { type: "string", default: "south" },
  },
  init() {
    this.el.addEventListener("model-loaded", () => {
      const windTurbineMesh = this.el.getObject3D("mesh");
      this.positionMesh(windTurbineMesh);
      const towerMesh = windTurbineMesh.children.find(
      (element) => element.name === "tower"
      );
      this.setupTower(towerMesh);
      windTurbineMesh.children.forEach((element) => {
        this.setupRotorHeight(element);
        this.setupBlades(element);
      });
      this.el.isWindModelLoaded = true;
    });
  },
  setupBlades(element) {
    const bladeRegExp = new RegExp("blade.*$");
    if (bladeRegExp.test(element.name)) {
      element.scale.x = this.data.rotorScale;
      element.position.y += 2;
      this.data._windTurbineBlades
      ? this.data._windTurbineBlades.push(element)
      : (this.data._windTurbineBlades = [element]);
    }
  },
  setupTower(element) {
    element.scale.z = this.data.heightScale;
    const size = new window.THREE.Box3().setFromObject(element).
        getSize();
    this.el._windTurbineRotorYPosition = size.y;
  },
  setupRotorHeight(element) {
    if (element.name !== "tower") {
      element.position.y = this.el._windTurbineRotorYPosition;
    }
  },
  positionMesh(windTurbine) {
    switch (this.data.direction) {
      case "south":
      return windTurbine.rotation.set(0, 0, 0);
      case "north":
      return windTurbine.rotation.set(0, (180 * Math.PI) / 180, 0);
      case "east":
      return windTurbine.rotation.set(0, (90 * Math.PI) / 180, 0);
      case "west":
      return windTurbine.rotation.set(0, (-90 * Math.PI) / 180, 0);
      case "northwest":
```

```
54        return windTurbine.rotation.set(0, (-135 * Math.PI) / 180, 0);
55        case "northeast":
56        return windTurbine.rotation.set(0, (135 * Math.PI) / 180, 0);
57        case "southeast":
58        return windTurbine.rotation.set(0, (45 * Math.PI) / 180, 0);
59        case "southwest":
60        return windTurbine.rotation.set(0, (-45 * Math.PI) / 180, 0);
61      }
62    },
63  });
```

Listing 4. Schema and setup to register a stationary wind turbine component

B. Code of Implementation

```
function waitForVideo() {
  dispatchCurrentState(false, "Waiting for video...");
  const observer = new MutationObserver((mutations, obs) => {
    const video = document.getElementsByTagName("video")[0];
    if (video) {
      console.log("video loaded");
      obs.disconnect();
      arjsVideo = video;
      video.addEventListener("loadedMetadata", findVideoPosition());
    }
  });
  observer.observe(document, {
    childList: true,
    subtree: true,
  });
}
```

Listing 5. Implementation of waitForVideo()

```
function findVideoPosition() {
  dispatchCurrentState(false, "Finding video position...");
  return new Promise((resolve, reject) => {
    if (
    JSON.stringify(arjsVideo.getBoundingClientRect()) ===
    JSON.stringify(arjsVideoRect)
    ) {
      // now the video is set correctly.
      videoPositioned = true;
      loadNeuronalNetwork();
      loadWindturbines();
      placeOverlay();
    } else {
      // video not yet positioned correctly
      arjsVideoRect = arjsVideo.getBoundingClientRect();
      setTimeout(findVideoPosition, 100);
    }
  });
}
```

Listing 6. Implementation of findVideoPosition()

```
function loadNeuronalNetwork() {
  dispatchCurrentState(false, "Loading neural network...");
  // Load neural network here
}
```

Listing 7. Implementation of loadNeuronalNetwork()

```
async function loadWindturbines() {
  dispatchCurrentState(false, "Loading windturbine information...");
  await fetch("../windturbineDB.json")
```

```
  .then((response) => {
    console.log(response);
    return response.json();
  })
  .then((data) => {
    windturbines = data.windturbines;
    windturbineLocations = data.windturbines.map((windturbine) => {
      return [
      windturbine?.id,
      windturbine?.location?.latitude,
      windturbine?.location?.longitude,
      ];
    });
    console.error("winturbinelocations", windturbineLocations,
        windturbines);
    appendWindturbines();
  });
  toggleBladeRotation();
}
```

Listing 8. Implementation of loadWindturbines()

```
function appendWindturbines() {
  for (var i = 0; i < windturbines.length && i < numberOfWindturbines
      ; i++) {
    if (rotationToggle) {
      try {
        const windturbineElement = document.createElement("a-entity")
            ;

        windturbineElement.setAttribute(
        "id",
        `windturbineElement-${windturbines[i].id}`
        );
        windturbineElement.setAttribute(
        "gltf-model",
        windturbines[i].entity.model
        );
        windturbineElement.setAttribute(
        "gps-entity-place",
        `latitude: ${windturbines[i].location.latitude}; longitude: $
            {windturbines[i].location.longitude}`
        );
        windturbineElement.setAttribute(
        "windturbine-rotating",
        "speed: " +
        windturbines[i].entity.speed +
        "; heightScale: " +
        windturbines[i].entity.heightScale +
        "; rotorScale: " +
        windturbines[i].entity.rotorScale +
        "; direction: " +
        windturbines[i].entity.direction +
```

```
        ";"
        );
        appendCounterpart(windturbineElement);
      } catch (error) {
        console.error("Appending rotating windturbine failed: " +
            error);
      }
    } else {
      try {
        const windturbineElement = document.createElement("a-entity")
            ;
        windturbineElement.setAttribute(
        "id",
        `windturbineElement-${windturbines[i].id}`
        );
        windturbineElement.setAttribute(
        "gltf-model",
        windturbines[i].entity.model
        );
        windturbineElement.setAttribute(
        "gps-entity-place",
        `latitude: ${windturbines[i].location.latitude}; longitude: $
            {windturbines[i].location.longitude}`
        );
        windturbineElement.setAttribute(
        "windturbine-still",
        "speed: " +
        windturbines[i].entity.speed +
        "; heightScale: " +
        windturbines[i].entity.heightScale +
        "; rotorScale: " +
        windturbines[i].entity.rotorScale +
        "; direction: " +
        windturbines[i].entity.direction +
        ";"
        );
        appendCounterpart(windturbineElement);
      } catch (error) {
        console.error("Appending rotating windturbine failed: " +
            error);
      }
    }
  }
}
```

Listing 9. Implementation of appendWindturbines()

```
function appendCounterpart(newElement) {
  const identifier = newElement.getAttribute("id");
  const id = identifier.split("-")[1];
  var referenceNode;
  if (identifier.includes("windturbineElement")) {
    try {
```

```
      referenceNode = document.getElementById(`overlayCanvas-${id}`);
    } catch (error) {
      console.error("Appending after referenceNode failed: " + error);
    }
    if (referenceNode) {
      insertAfter(newElement, referenceNode);
    }
  } else {
    try {
      referenceNode = document.getElementById(`windturbineElement-${id}`);
    } catch (error) {
      console.error("Appending after referenceNode failed: " + error);
    }
    if (referenceNode) {
      referenceNode.parentNode.insertBefore(newElement, referenceNode);
    }
  }
}
```

Listing 10. Implementation of appendCounterpart()

```
function insertAfter(newNode, referenceNode) {
  referenceNode.parentNode.insertBefore(newNode, referenceNode.nextSibling);
}
```

Listing 11. Implementation of insertAfter()

```
function toggleBladeRotation() {
  if (rotationToggle === rotationToggleNewState) return;
  deleteWindturbines();
  rotationToggle = rotationToggleNewState;
  appendWindturbines();
}
```

Listing 12. Implementation of toggleBladeRotation()

```
function deleteWindturbines() {
  console.log("entered delete Windturbines");
  let entities = document.getElementsByTagName("a-entity");
  entities = Object.values(entities).filter(
  (entity) =>
  entity.hasAttribute("windturbine-rotating") ||
  entity.hasAttribute("windturbine-still")
  );
  console.log("entities: " + entities.length);
  if (!entities) return;
  for (var i = 0; i < entities.length; i++) {
    console.log("entered for loop");
```

```
13      if (
14      entities[i].hasAttribute("windturbine-rotating") ||
15      entities[i].hasAttribute("windturbine-still")
16      ) {
17        console.log("Entered deletion");
18        entities[i].parentNode.removeChild(entities[i]);
19      }
20    }
21  }
```

Listing 13. Implementation of deleteWindturbines()

```
1  function placeOverlay() {
2    dispatchCurrentState(false, "Placing overlay canvas...");
3
4    asceneElement = document.getElementsByTagName("a-scene")[0];
5    asceneElement.style.zIndex = "0";
6
7    overlayCanvas = document.createElement("canvas");
8    overlayCanvas.setAttribute("id", "overlayCanvas");
9    overlayCanvas.width = screenWidth;
10   overlayCanvas.height = screenHeight;
11
12   overlayContext = overlayCanvas.getContext("2d");
13   overlayContext.fillRect(0, 0, screenWidth, screenHeight);
14   overlayContext.drawImage(arjsVideo, 0, 0, screenWidth, screenHeight
       );
15
16   overlayCanvas.style.position = "absolute";
17   overlayCanvas.style.top = 0 + "px";
18   overlayCanvas.style.right = 0 + "px";
19   overlayCanvas.style.left = 0 + "px";
20   overlayCanvas.style.bottom = 0 + "px";
21
22   overlayCanvas.style.zIndex = "1";
23   document.body.prepend(overlayCanvas);
24
25   videoWidth = Math.abs(arjsVideoRect.width) - (Math.abs(
       arjsVideoRect.width) % 4);
26   videoHeight = Math.abs(arjsVideoRect.height);
27
28   hiddenCanvas = document.createElement("canvas");
29   hiddenCanvas.width = videoWidth;
30   hiddenCanvas.height = videoHeight;
31   hiddenctx = hiddenCanvas.getContext("2d");
32
33   dispatchCurrentState(true, "Done loading");
34   if (videoRendering) {
35     loop();
36   }
37 }
```

Listing 14. Implementation of placeOverlay()

```
async function loop() {
  hiddenctx.clearRect(0, 0, videoWidth, videoHeight);
  hiddenctx.drawImage(arjsVideo, 0, 0, videoWidth, videoHeight);

  trimmedCanvasData = trimHiddenCanvasData(
    hiddenctx.getImageData(0, 0, videoWidth, videoHeight).data,
    videoWidth,
    videoHeight
  );

  distance = getDistance();
  calcDepthMap(trimmedCanvasData.imageData, distance);
  calcBinaryMap(depthmap, distance);
  applyBinaryMap(trimmedCanvasData.clampedArray, binaryMap);

  if (videoRendering) {
    setTimeout(loop, loopdelay);
  }
}
```

Listing 15. Implementation of loop()

```
function trimHiddenCanvasData(dataArray, videoWidth, videoHeight) {
  const totalOffset = videoWidth - screenWidth;
  if (totalOffset > 0) {
    offsetLeft = Math.floor(totalOffset / 2);
    offsetRight = Math.ceil(totalOffset / 2);
  }
  var pos = 0;
  const widthStart = offsetLeft * 4;
  const widthEnd = (videoWidth - offsetRight) * 4;
  const valuesPerWidth = videoWidth * 4;
  for (var i = 0; i < videoHeight; i++) {
    for (var j = widthStart; j < widthEnd; j++) {
      resizedArray[pos] = dataArray[i * valuesPerWidth + j];
      pos++;
    }
  }
  const imageData = new ImageData(resizedArray, screenWidth);
  return {
    clampedArray: resizedArray,
    imageData: imageData,
  };
}
```

Listing 16. Implementation of trimHiddenCanvasData()

```
function getDistance() {
  function calcDistance(userPosition) {
    if (!userPosition) return;
    var gpsEntities = document.getElementsByTagName("a-entity");
    const gpsEntitiesCopy = [];
    for (var i = 0; i < gpsEntities.length; i++) {
```

```
7       if (gpsEntities[i].hasAttribute("gltf-model")) {
8         gpsEntitiesCopy.push(gpsEntities[i]);
9       }
10    }
11    gpsEntities = gpsEntitiesCopy;
12    if (!gpsEntities) return;
13    let windturbineCoords = [];
14    for (var i = 0; i < gpsEntities.length; i++) {
15      windturbineCoords.push(gpsEntities[i].getAttribute("gps-entity-
          place"));
16    }
17    var userCoords = userPosition.coords;
18    for (var i = 0; i < numberOfWindturbines; i++) {
19      distanceBetweenUserAndWindTurbine[i] = calcCrow(
20      userCoords.latitude,
21      userCoords.longitude,
22      windturbineCoords[i].latitude,
23      windturbineCoords[i].longitude
24      );
25    }
26  }
27
28  function calcCrow(lat1, lon1, lat2, lon2) {
29    var R = 6_371_000; // in meter
30    var lat1 = toRad(lat1);
31    var lat2 = toRad(lat2);
32    var dLat = toRad(lat2 - lat1);
33    var dLon = toRad(lon2 - lon1);
34
35    var a =
36    Math.sin(dLat / 2) * Math.sin(dLat / 2) +
37    Math.sin(dLon / 2) * Math.sin(dLon / 2) * Math.cos(lat1) * Math.
        cos(lat2);
38    var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
39    var d = R * c;
40    return d.toFixed(2);
41  }
42
43  // Converts numeric degrees to radians
44  function toRad(value) {
45    return (value * Math.PI) / 180;
46  }
47
48  if (navigator.geolocation) {
49    navigator.geolocation.getCurrentPosition(
50    calcDistance,
51    locationError,
52    gpsOptions
53    );
54    return distanceBetweenUserAndWindTurbine;
55  } else {
56    console.log("no GPS available");
```

```
  }
  function locationError(err) {
    console.warn(`ERROR(${err.code}): ${err.message}`);
  }
}
```

Listing 17. Implementation of getDistance()

```
function calcDepthMap(imageData, distance) {
  /**
  * this function does not need the second parameter distance, yet the calculation of the distance
  * works, but the calc depth is not correctly implemented since the neural network is missing.
  * the attribute distance is here used to simulate the behavior of the neural network
  */
  const totalPixels = screenWidth * screenHeight;
  for (
  var tophalfpixel = 0;
  tophalfpixel < Math.ceil(totalPixels / 3);
  tophalfpixel++
  ) {
    depthmap[tophalfpixel] = Math.floor((Math.random() + 1) * 100000);
  }
  for (var windturbine = 0; windturbine < numberOfWindturbines; windturbine++) {
    for (
    var j =
    Math.ceil(totalPixels / 3) +
    Math.floor((2 * totalPixels) / 3 / numberOfWindturbines) * windturbine;
    j <
    Math.ceil(totalPixels / 3) +
    Math.floor((2 * totalPixels) / 3 / numberOfWindturbines) *
    (windturbine + 1);
    j++
    ) {
      depthmap[j] =
      (Math.random() + 1) * 2 * (Math.round(Math.random()) ? 1 : -1) +
      Math.floor(distance[windturbine]);
    }
  }
}
```

Listing 18. Implementation of calcDepthMap()

```
function calcBinaryMap(depthmap, distance) {
  for (var windturbine = 0; windturbine < numberOfWindturbines; windturbine++) {
    for (var depthpixel = 0; depthpixel < depthmap.length; depthpixel++) {
```

```
      binaryMap[windturbine][depthpixel] =
      depthmap[depthpixel] < distance[windturbine] ? 1 : 0;
    }
  }
}
```

Listing 19. Implementation of calcBinaryMap()

```
function applyBinaryMap(clampedArray, binaryMap) {
  var binaryCounter = 0;
  var clampedCounter = 3;
  var allClampedArrays = [];
  for (var i = 0; i < numberOfWindturbines; i++) {
    allClampedArrays.push(clampedArray);
  }
  for (var i = 0; i < numberOfWindturbines; i++) {
    while (binaryCounter < binaryMap[i].length) {
      if (binaryMap[i][binaryCounter] != 1) {
        allClampedArrays[i][clampedCounter] = 0;
      }
      binaryCounter++;
      clampedCounter = clampedCounter + 4;
    }
    binaryCounter = 0;
    clampedCounter = 3;
  }
  for (let i = 0; i < numberOfWindturbines; i++) {
    var imageData;
    try {
      imageData = new ImageData(allClampedArrays[i], screenWidth);
    } catch (error) {
      dispatchCurrentState(false, "An error occured, please restart
          the app!");
    }
    overlayContext[i].clearRect(0, 0, screenWidth, screenHeight);
    overlayContext[i].putImageData(imageData, 0, 0);
  }
}
```

Listing 20. Implementation of applyBinaryMap()

C. Code of Screenshot Implementations

```
1 import { Screenshot } from "@ionic-native/screenshot";
2 function takeScreenshot() {
3   Screenshot.save("jpg", 80, "myscreenshot.jpg");
4 }
```

Listing 21. Implementation of official Capacitor Screenshot plugin

```
1 async function takeScreenshot() {
2   const stream = await navigator.mediaDevices.getDisplayMedia();
3   const screenVideoTrack = stream.getVideoTracks()[0];
4   const capture = new ImageCapture(screenVideoTrack);
5 }
```

Listing 22. Implementation of Screen Capture API

D. Test results

Videos recorded with a tripod

Video	Model	Average	Average Percentage	Minimum	Maximum	Difference Min Max	% Difference over Average
1	small	595,221	28.7%	591,760	598,604	6844	1.15%
1	medium	603,013	29.08%	601,765	605,097	3332	0.55%
1	large	606,470	29.25%	604,660	608,431	3771	0.62%
2	small	183,495	8.85%	177,008	193,397	16,389	8.93%
2	medium	195,216	9.41%	189,701	202,732	13,031	6.68%
2	large	198,681	9.58%	188,214	209,459	21,245	10.69%
3	small	678,855	32.74%	661,120	694,594	33,474	4.93%
3	medium	673,532	32.48%	670,487	677,979	7492	1.11%
3	large	685,664	33.07%	681,580	689,916	8336	1.22%
4	small	195,670	9.44%	147,177	252,138	104,961	53.64%
4	medium	248,208	11.97%	243,580	256,232	12,652	5.1%
4	large	287,906	13.88%	278,958	298,370	19,412	6.74%
5	small	6662	0.32%	0	120,177	120,177	1803.92%
5	medium	55,624	2.68%	45,711	75,727	30,016	53.96%
5	large	127,448	6.15%	109,821	153,399	43,578	34.19%
6	small	198,906	9.59%	5573	322,372	316,799	159.27%
6	medium	222,963	10.75%	216,145	230,839	14,694	6.59%
6	large	246,969	11.91%	238,742	260,352	21,610	8.75%
7	small	549,767	26.51%	546,283	553,689	7406	1.35%
7	medium	566,144	27.3%	564,695	569,455	4760	0.84%
7	large	577,403	27.85%	574,929	582,047	7118	1.23%
8	small	158,956	7.67%	139,983	183,435	43,452	27.34%
8	medium	172,590	8.32%	165,396	178,621	13,225	7.66%
8	large	211,556	10.2%	205,559	220,311	14,752	6.97%
9	small	544,726	26.27%	515,044	623,862	108,818	19.98%
9	medium	557,632	26.89%	551,301	564,353	13,052	2.34%
9	large	568,642	27.42%	562,045	576,543	14,498	2.55%
10	small	389,940	18.8%	383,453	394,689	11,236	2.88%
10	medium	396,255	19.11%	393,931	398,586	4655	1.17%
10	large	406,428	19.6%	402,443	412,250	9807	2.41%

Table 6. Information about number of incalculable values per video and MiDaS model

Videos recorded without a tripod

Video	Model	Average	Average Percentage	Minimum	Maximum	Difference Min Max	% Difference over Average
1	small	463,326	22.34%	440,842	481,200	40,358	8.71%
1	medium	472,223	22.77%	454,439	487,170	32,731	6.93%
1	large	477,840	23.04%	413,725	498,005	84,280	17.64%
2	small	533,715	25.74%	500,102	575,066	74,964	14.05%
2	medium	539,524	26.02%	506,320	577,684	71,364	13.23%
2	large	537,915	25.94%	404,969	578,121	173,152	32.19%
3	small	311,220	15.01%	232,607	409,251	176,644	56.76%
3	medium	324,202	15.63%	246,738	420,814	174,076	53.69%
3	large	334,741	16.14%	255,769	429,625	173,856	51.94%
4	small	31,171	1.5%	0	64,988	64,988	208.49%
4	medium	77,206	3.72%	72,823	87,225	14,402	18.65%
4	large	86,607	4.18%	80,542	98,450	17,908	20.68%
5	small	566,176	27.3%	321,707	614,736	293,029	51.76%
5	medium	537,329	25.91%	134,400	626,681	492,281	91.62%
5	large	556,570	26.84%	370,618	634,006	263,388	47.32%
6	small	328,640	15.85%	251,637	411,115	159,478	48.53%
6	medium	330,114	15.92%	268,457	396,921	128,464	38.92%
6	large	308,349	14.87%	11,203	408,794	397,591	128.94%
7	small	552,720	26.66%	390,684	635,353	244,669	44.27%
7	medium	563,974	27.2%	399,943	645,034	245,091	43.46%
7	large	570,847	27.53%	406,535	653,906	247,371	43.33%
8	small	534,164	25.76%	464,934	645,227	180,293	33.75%
8	medium	542,136	26.14%	477,046	652,065	175,019	32.28%
8	large	541,836	26.13%	290,872	657,617	366,745	67.69%
9	small	632,841	30.52%	0	705,869	705,869	111.54%
9	medium	675,324	32.57%	579,854	722,975	143,121	21.19%
9	large	677,437	32.67%	582,671	725,417	142,746	21.07%
10	small	435,859	21.02%	280,117	552,929	272,812	62.59%
10	medium	467,589	22.55%	393,757	542,726	148,969	31.86%
10	large	502,165	24.22%	391,273	588,540	197,267	39.28%

Table 7. Information about number of incalculable values per video and MiDaS model

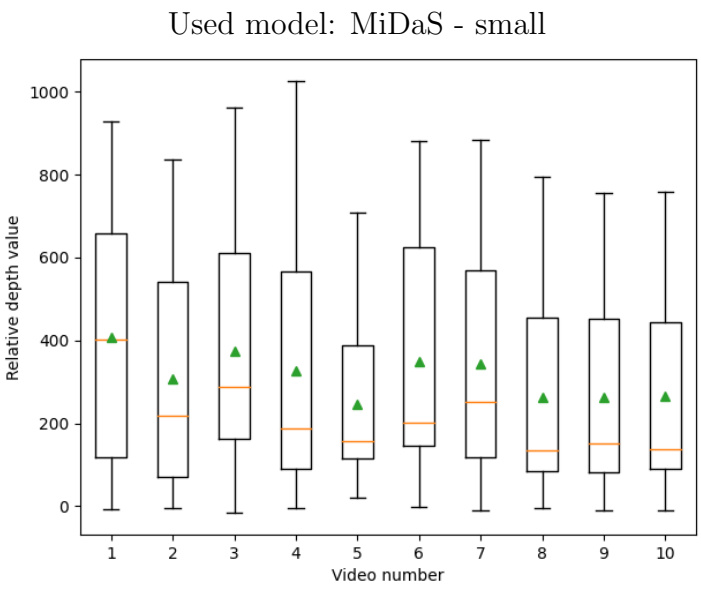

Figure 4. Distribution of average non-zero pixel values per video. Used model: small. Video recorded with a tripod.

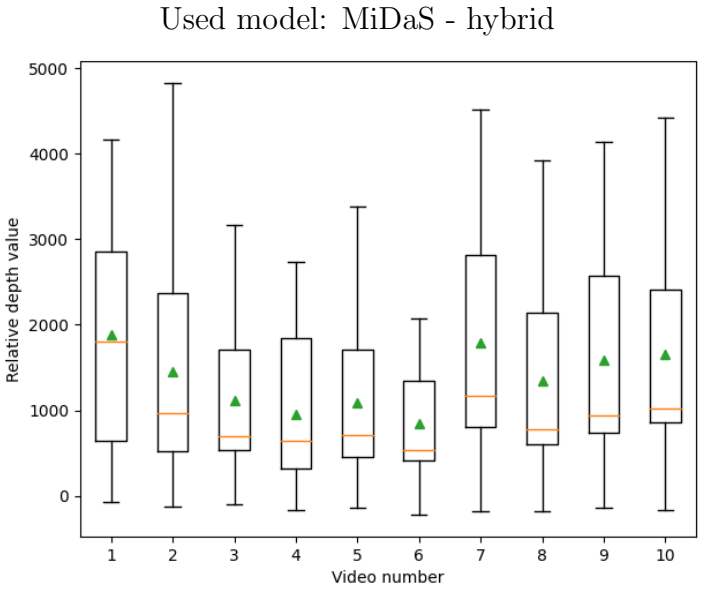

Figure 5. Distribution of average non-zero pixel values per video. Used model: hybrid. Video recorded with a tripod.

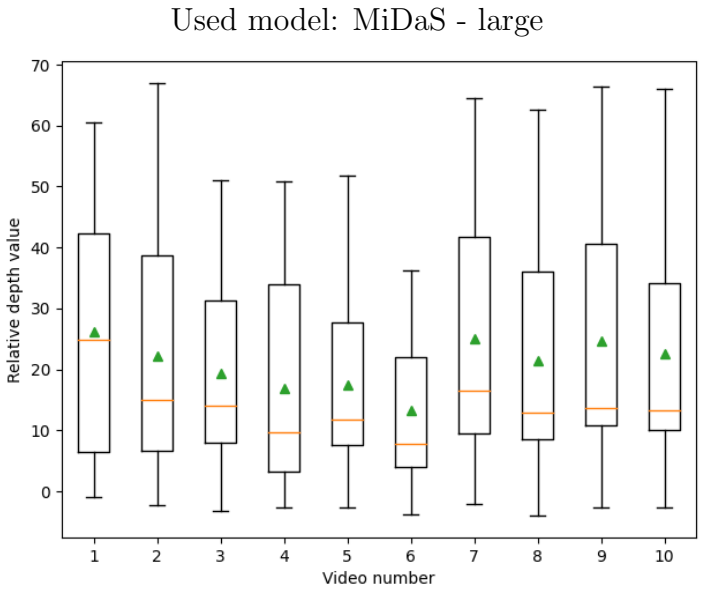

Figure 6. Distribution of average non-zero pixel values per video. Used model: large. Video recorded with a tripod.

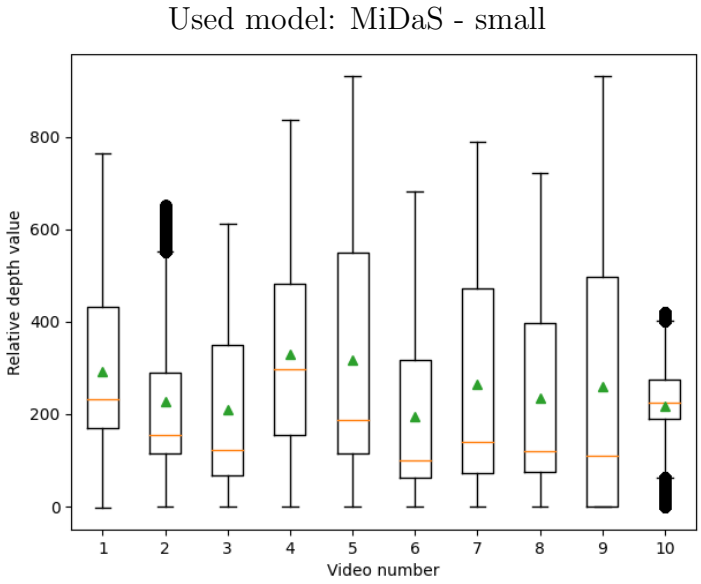

Figure 7. Distribution of average non-zero pixel values per video. Used model: small. Video recorded without a tripod.

Used model: MiDaS - hybrid

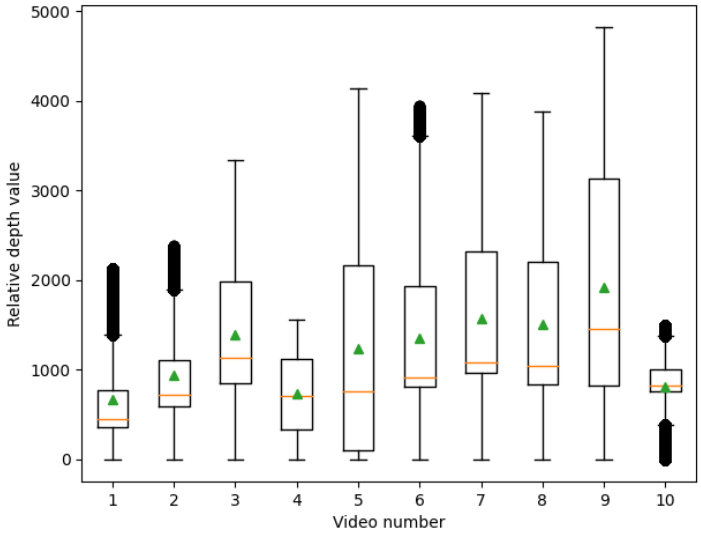

Figure 8. Distribution of average non-zero pixel values per video. Used model: hybrid. Video recorded without a tripod.

Used model: MiDaS - large

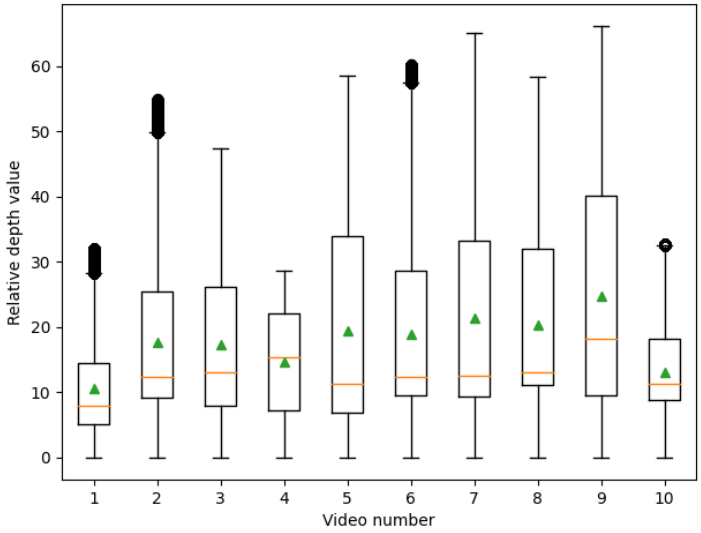

Figure 9. Distribution of average non-zero pixel values per video. Used model: large. Video recorded without a tripod.

Lowest and highest fluctuations of relative depth values per video and model

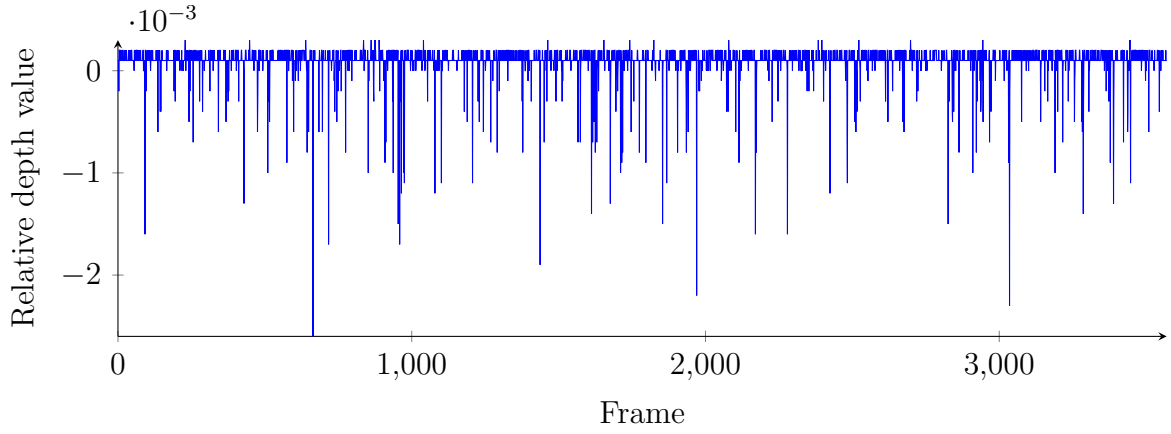

Figure 10. Lowest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-small

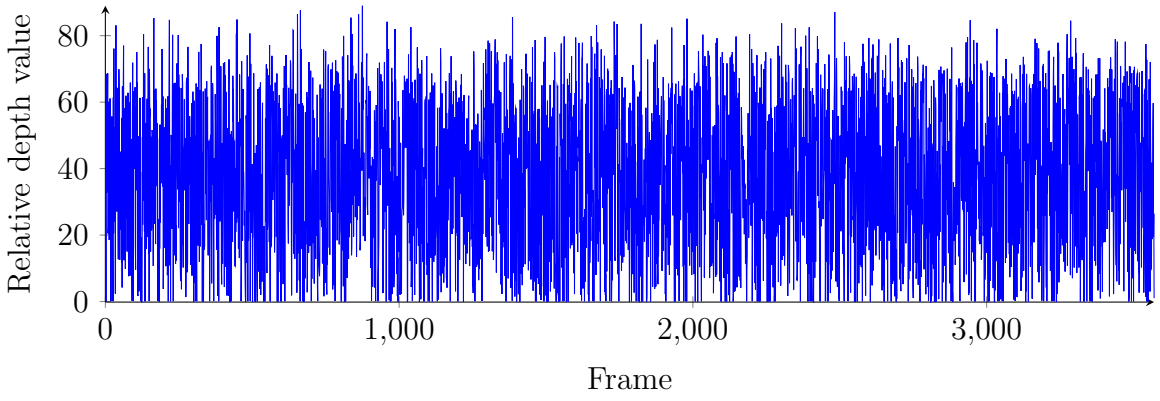

Figure 11. Highest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-small

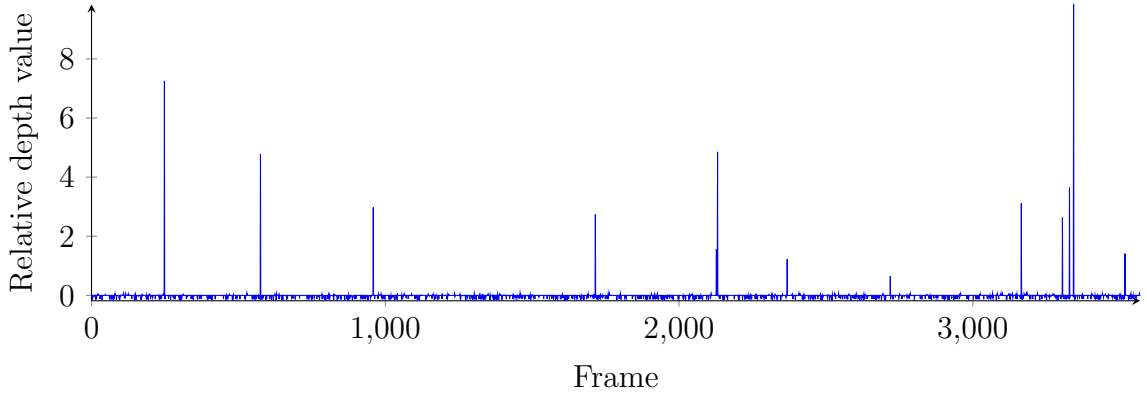

Figure 12. Lowest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-hybrid

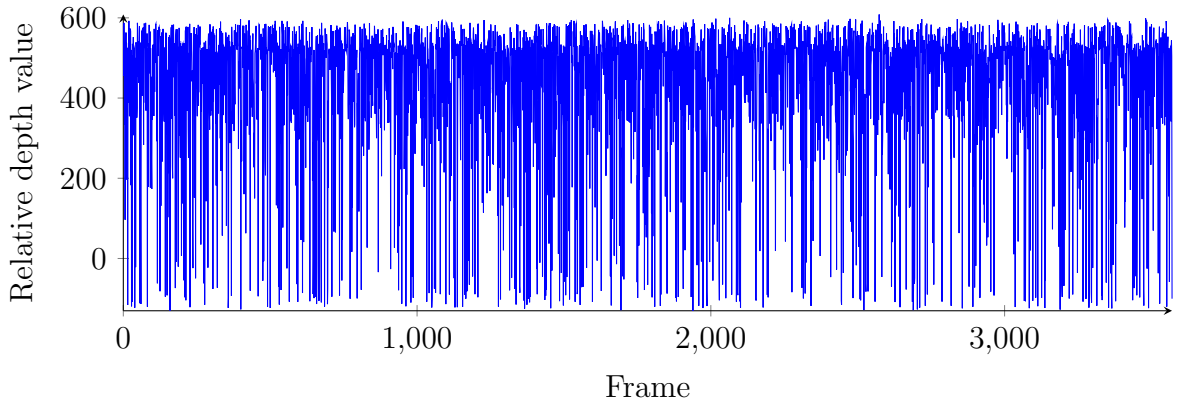

Figure 13. Highest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-hybrid

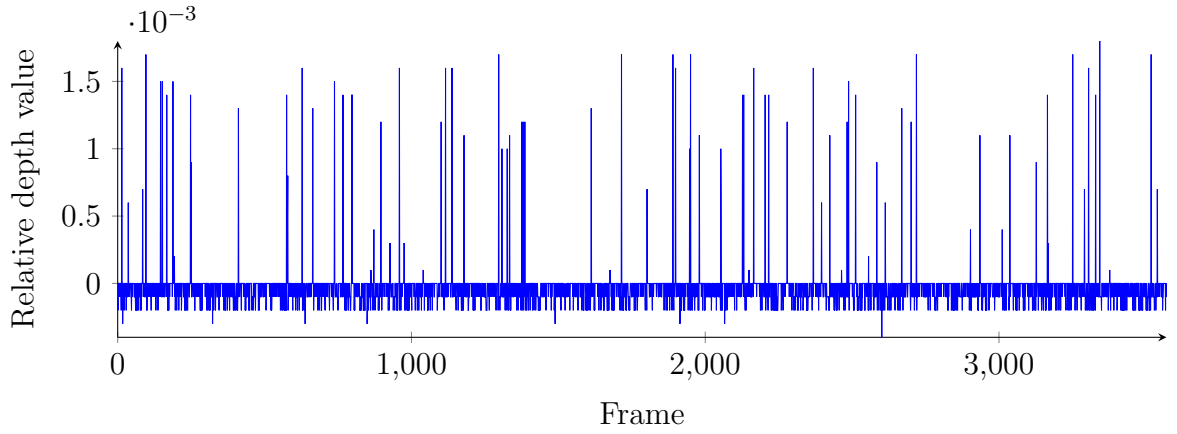

Figure 14. Lowest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-large

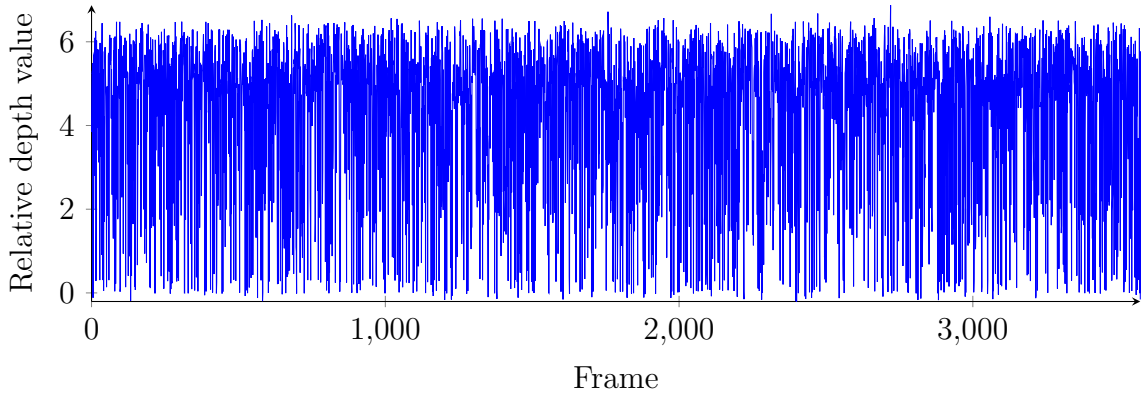

Figure 15. Highest variation of relative depth values.
Video used: Video 1 with tripod, Model used: MiDaS-large

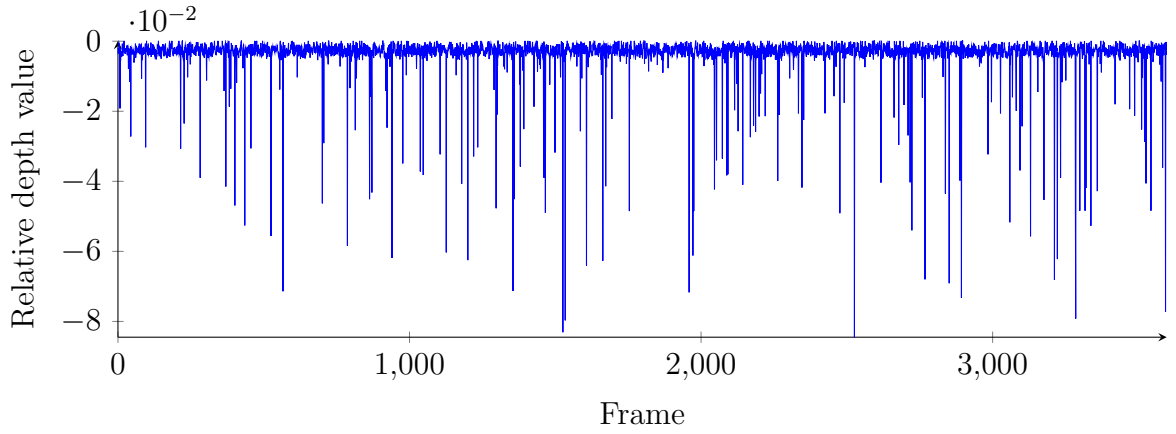

Figure 16. Lowest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-small

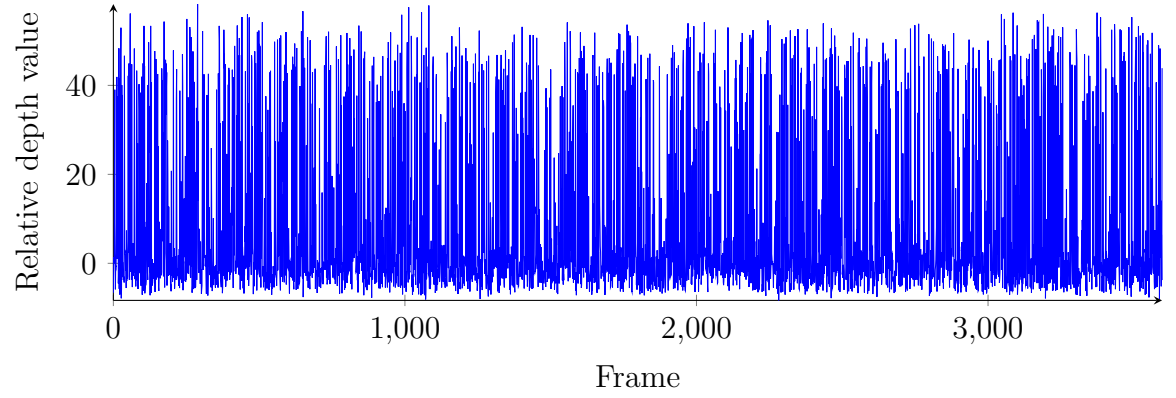

Figure 17. Highest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-small

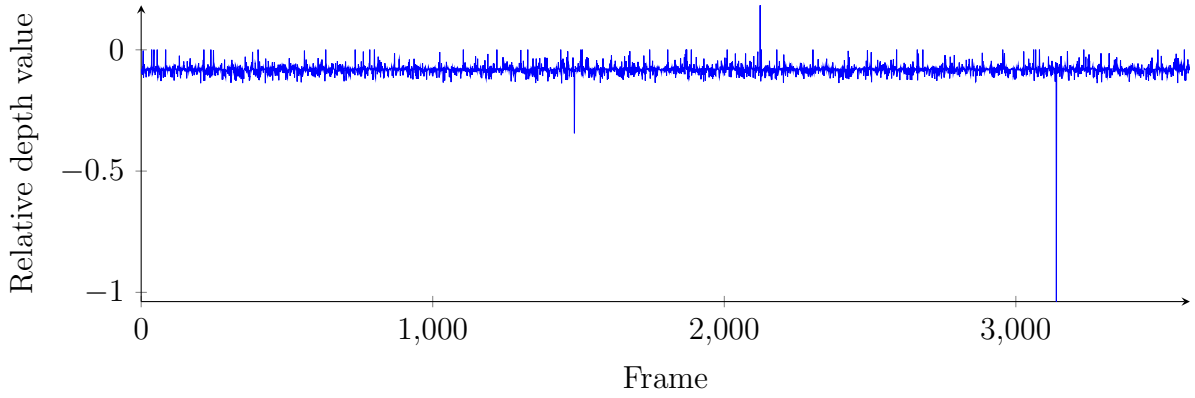

Figure 18. Lowest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-hybrid

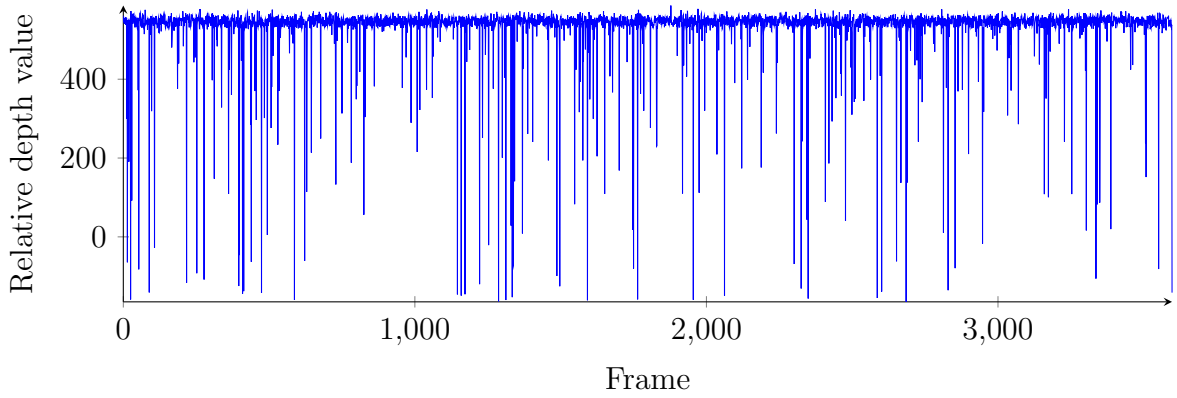

Figure 19. Highest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-hybrid

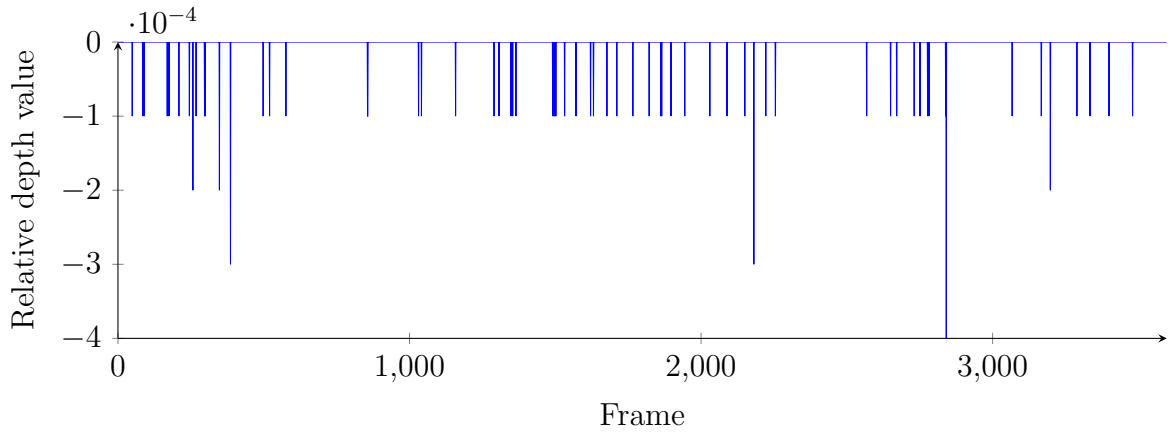

Figure 20. Lowest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-large

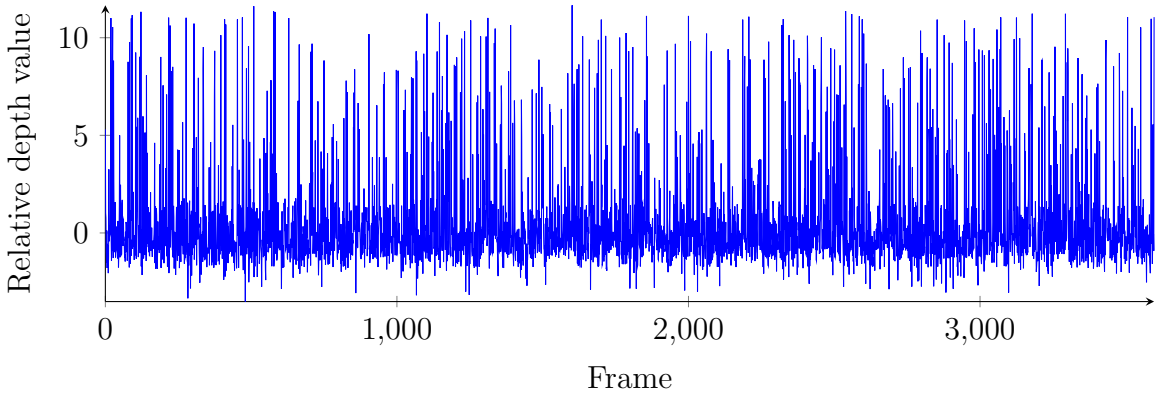

Figure 21. Highest variation of relative depth values.
Video used: Video 2 with tripod, Model used: MiDaS-large

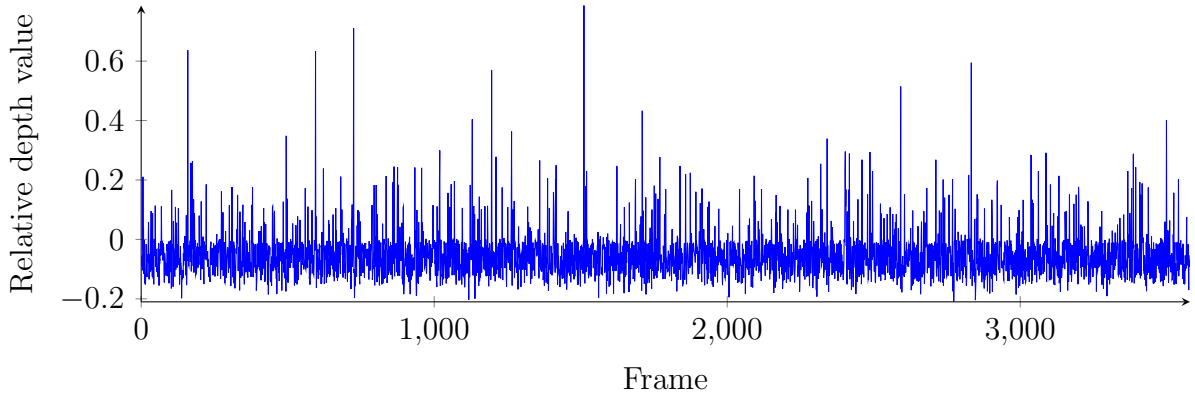

Figure 22. Lowest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-small

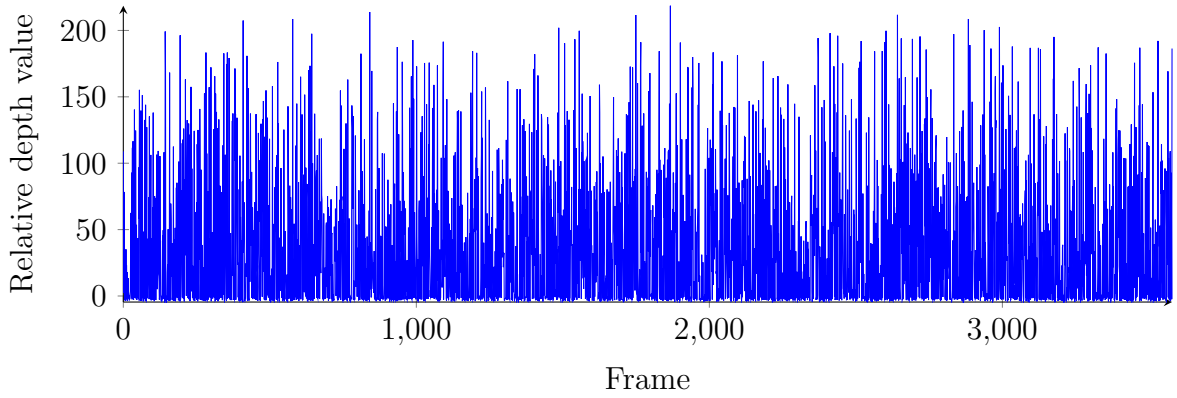

Figure 23. Highest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-small

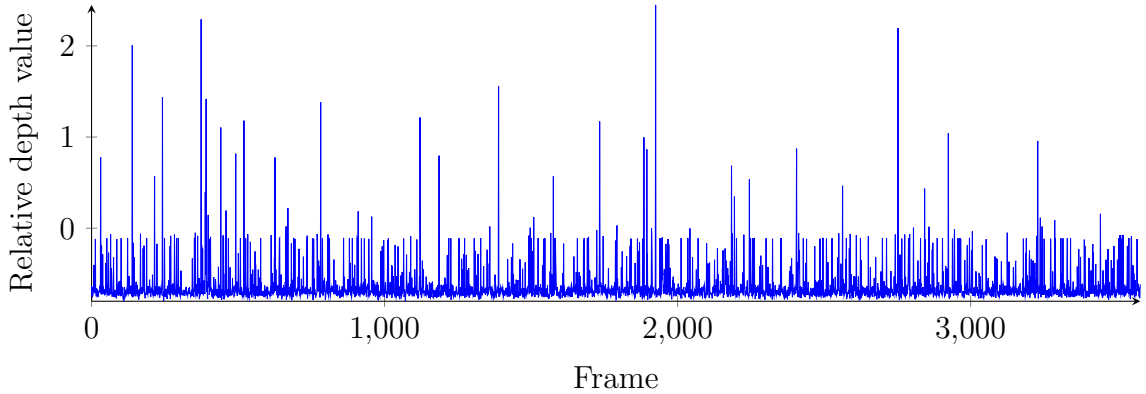

Figure 24. Lowest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-hybrid

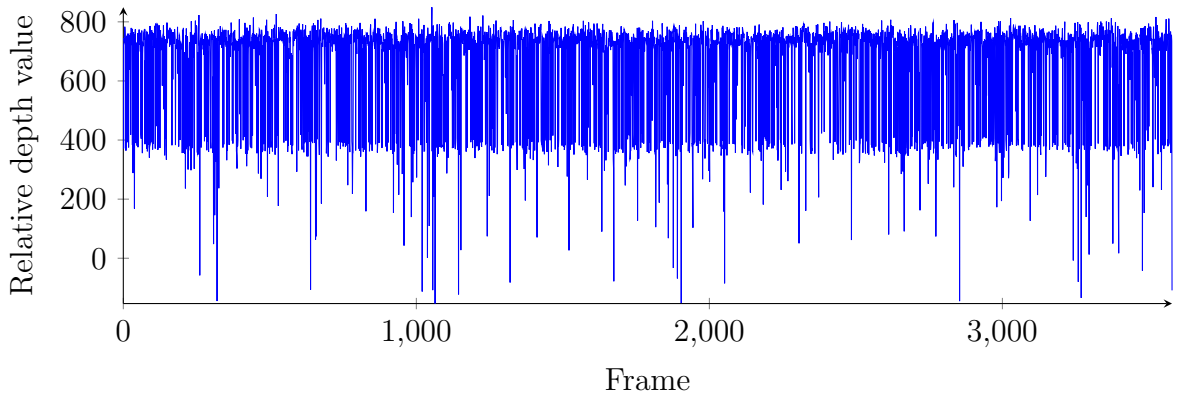

Figure 25. Highest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-hybrid

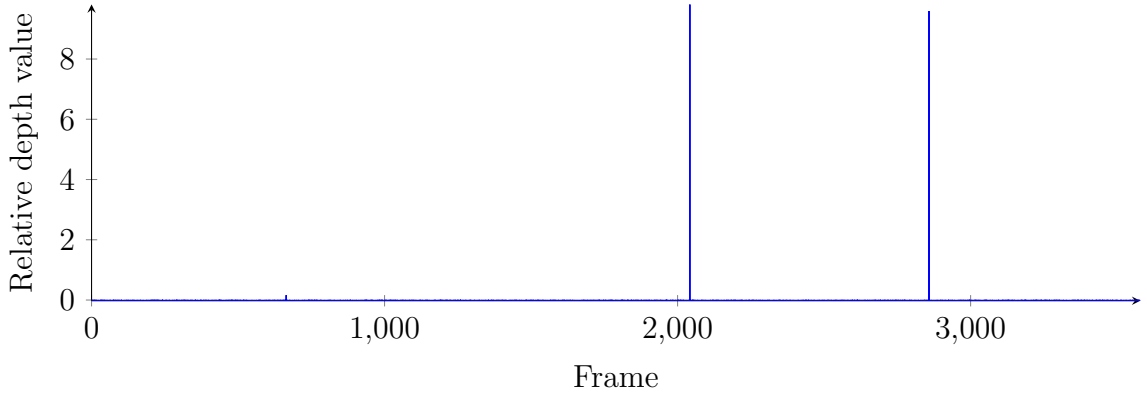

Figure 26. Lowest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-large

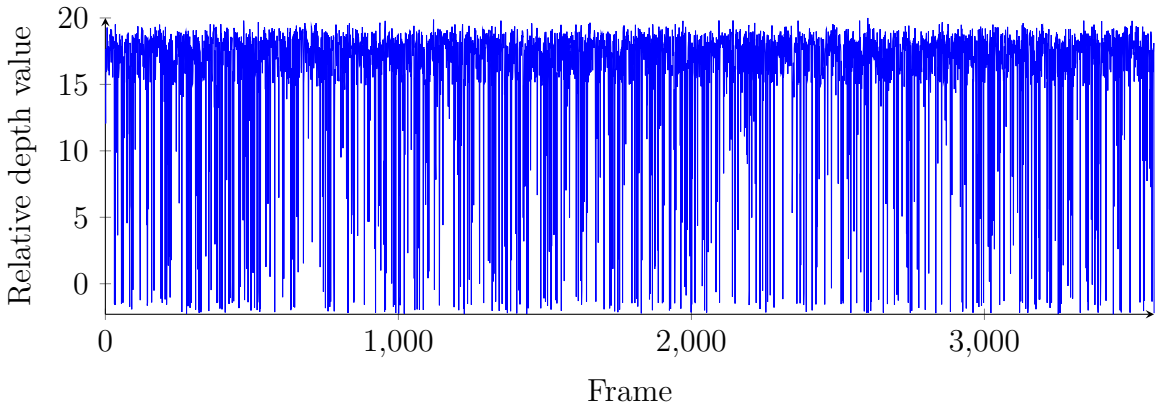

Figure 27. Highest variation of relative depth values.
Video used: Video 3 with tripod, Model used: MiDaS-large

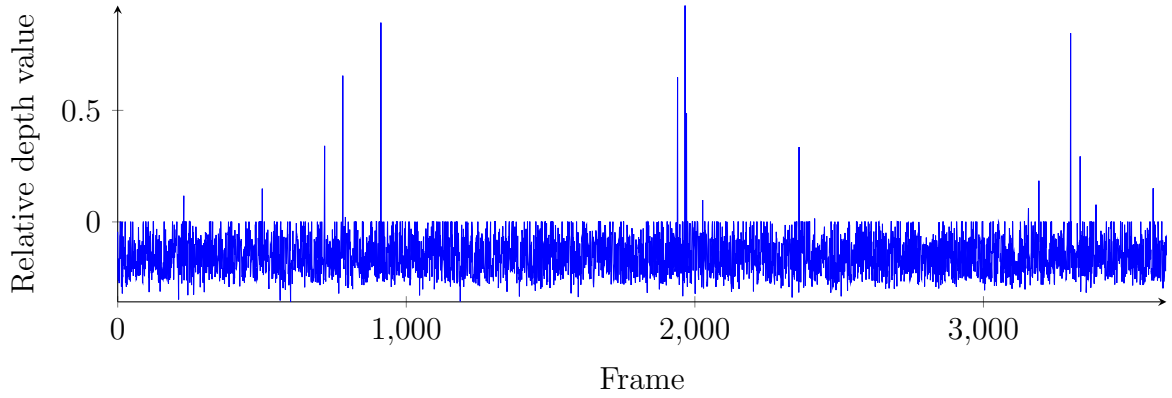

Figure 28. Lowest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-small

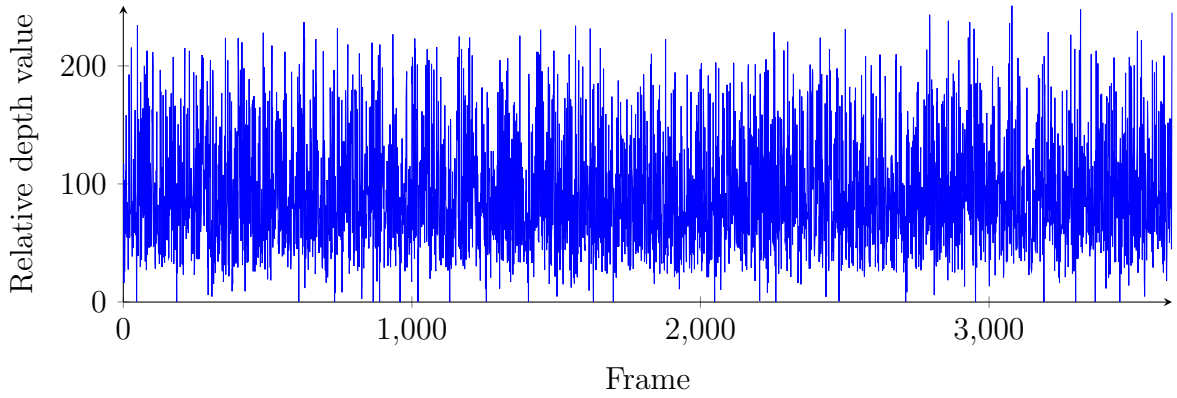

Figure 29. Highest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-small

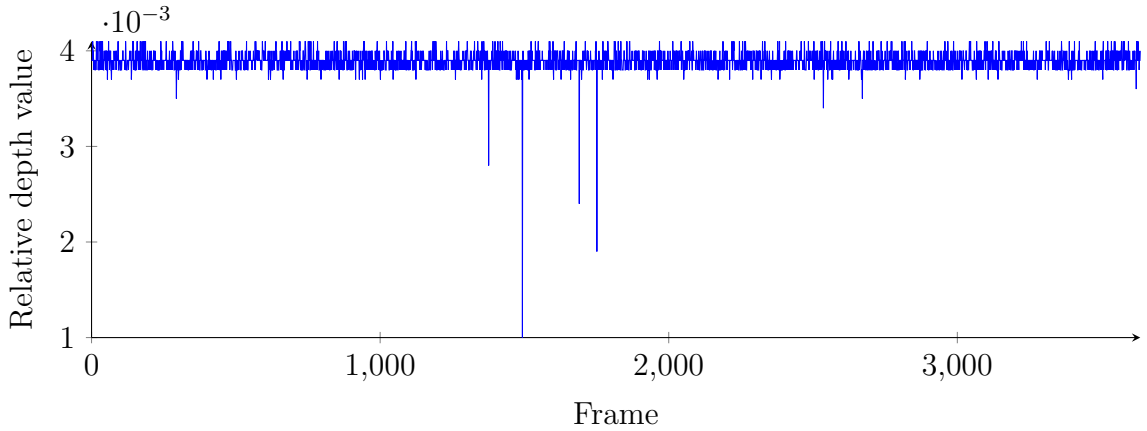

Figure 30. Lowest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-hybrid

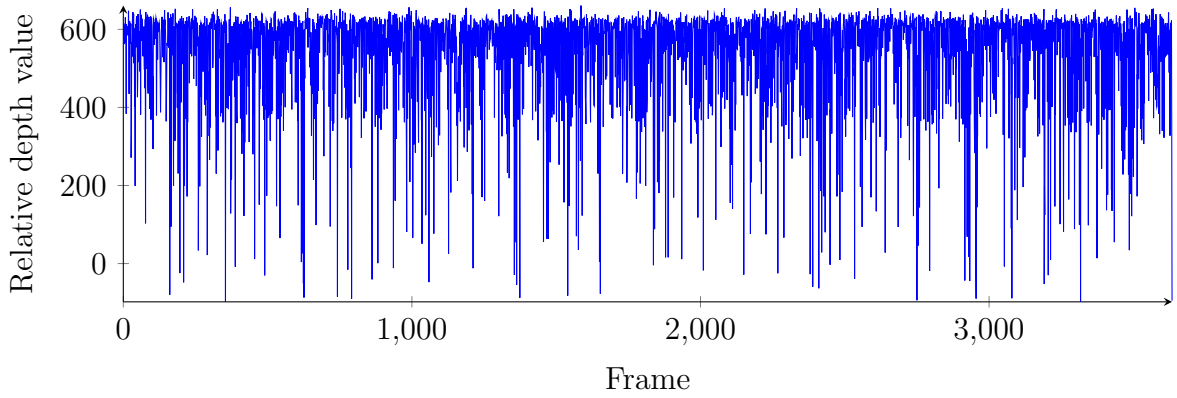

Figure 31. Highest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-hybrid

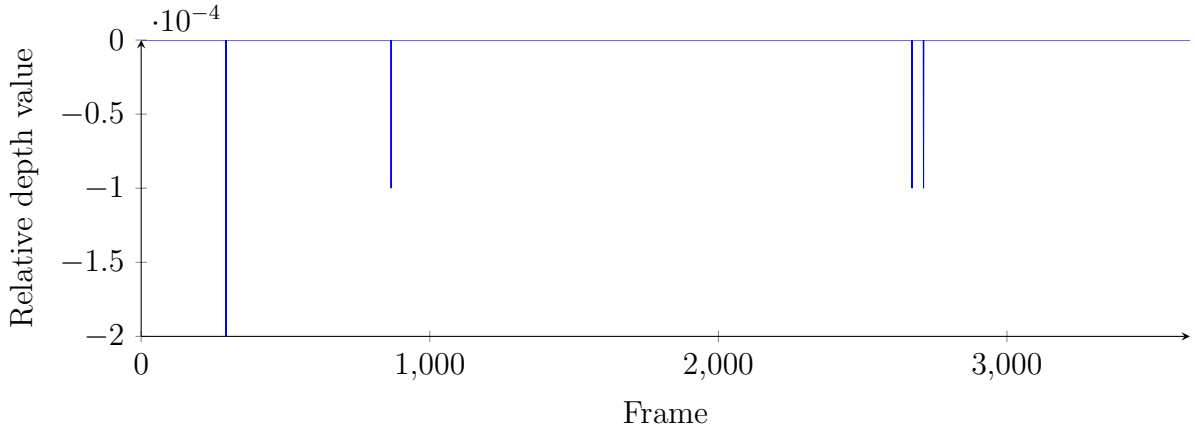

Figure 32. Lowest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-large

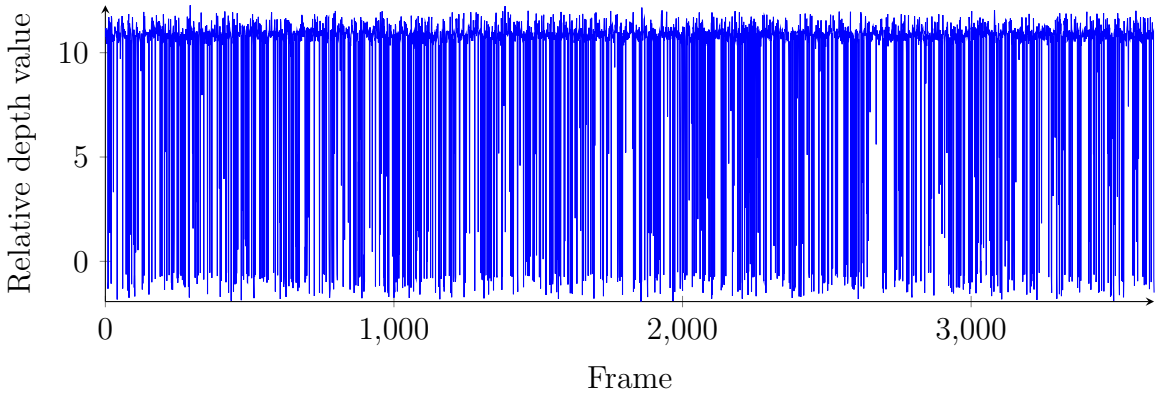

Figure 33. Highest variation of relative depth values.
Video used: Video 4 with tripod, Model used: MiDaS-large

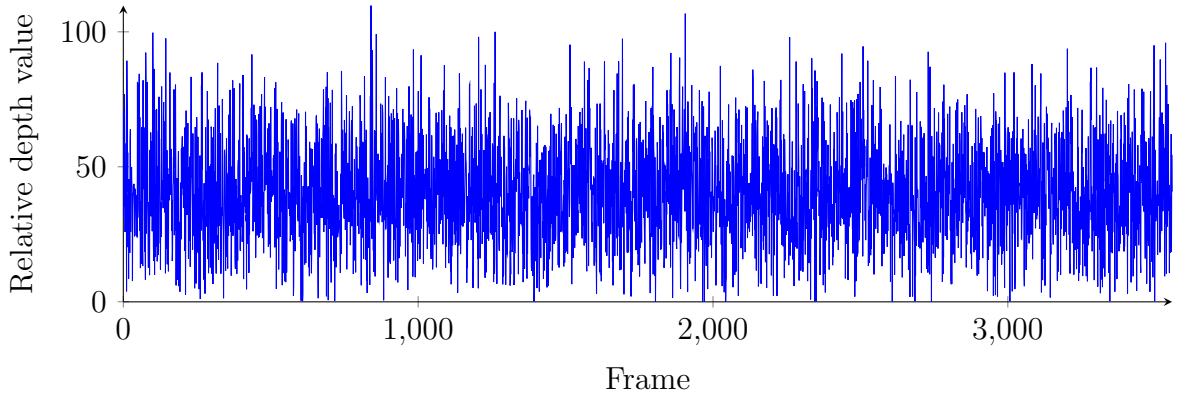

Figure 34. Lowest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-small

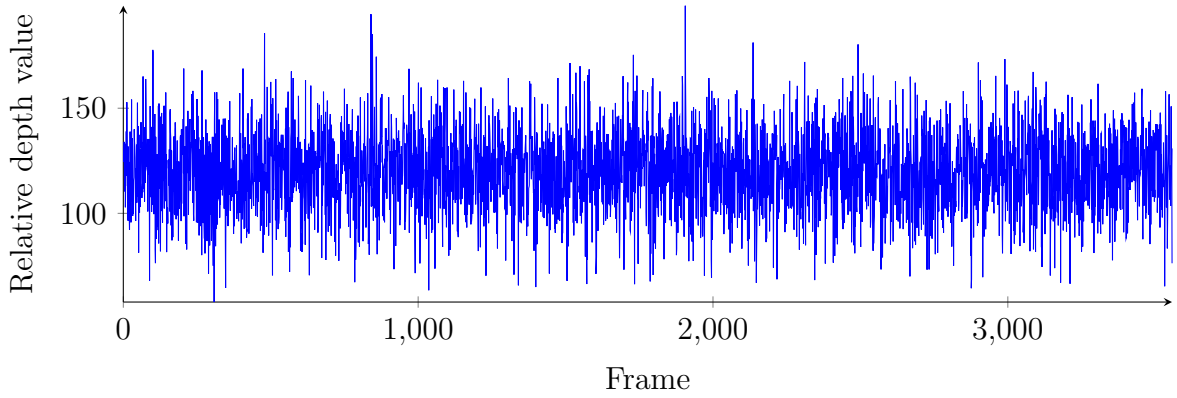

Figure 35. Highest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-small

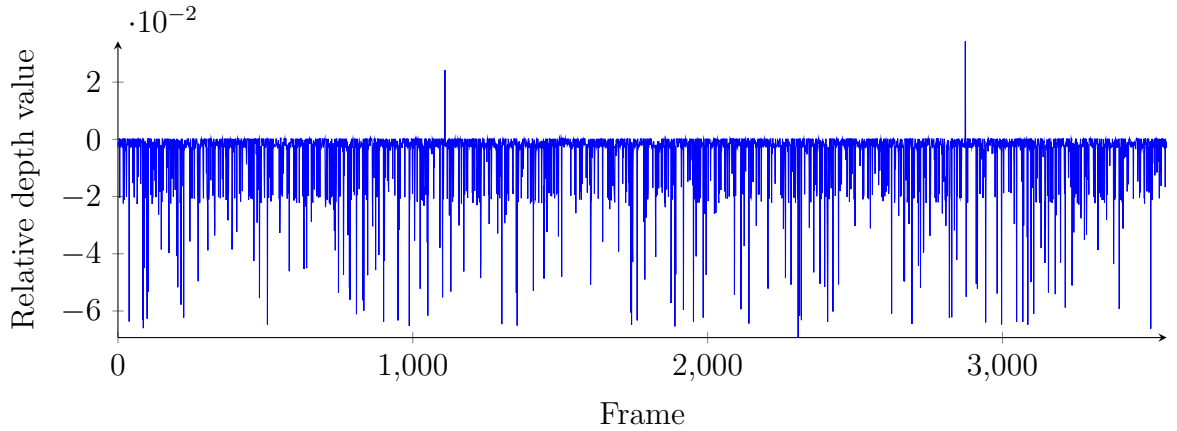

Figure 36. Lowest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-hybrid

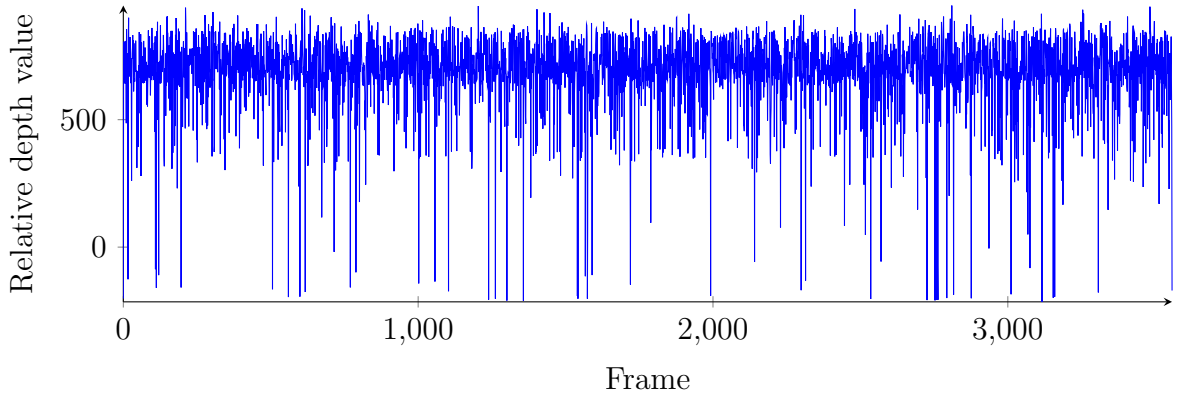

Figure 37. Highest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-hybrid

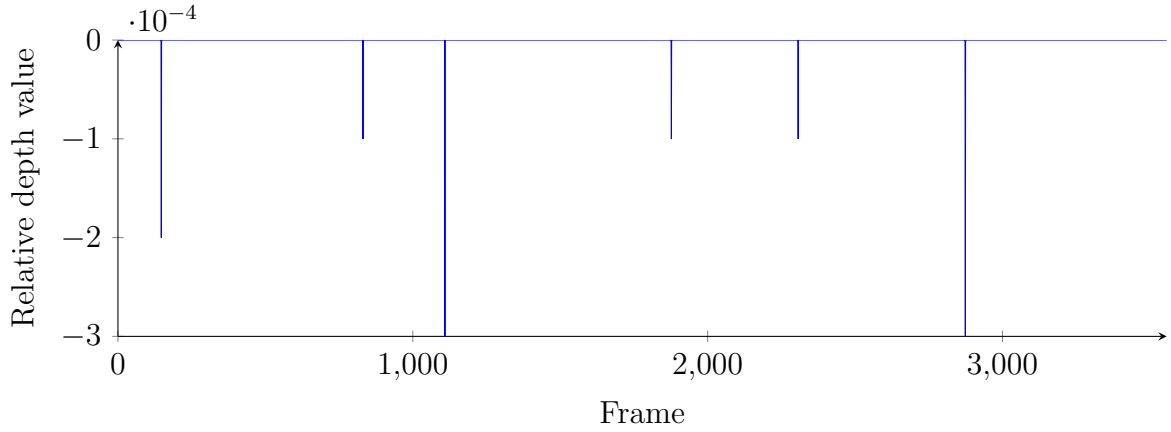

Figure 38. Lowest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-large

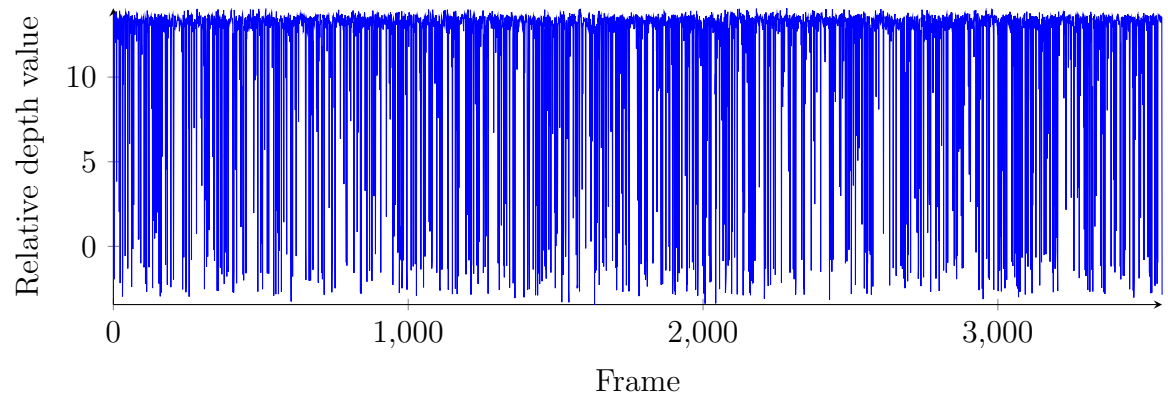

Figure 39. Highest variation of relative depth values.
Video used: Video 5 with tripod, Model used: MiDaS-large

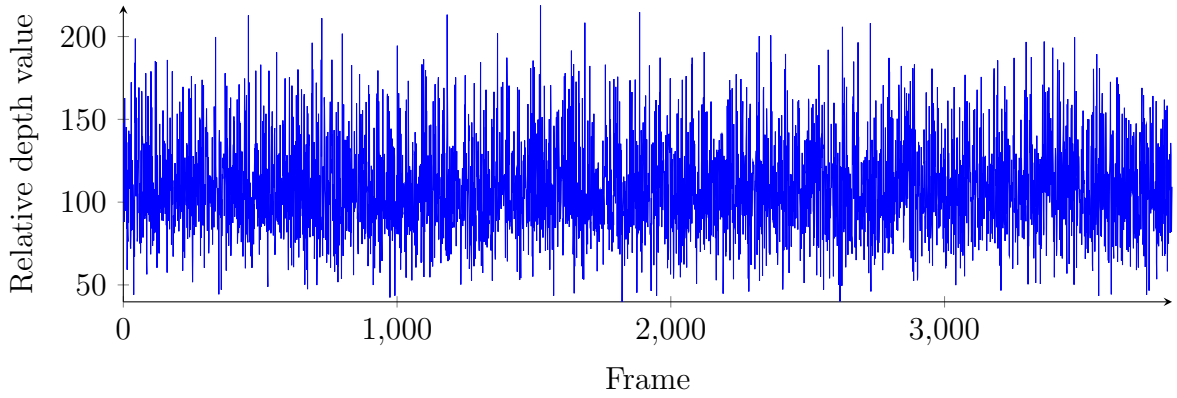

Figure 40. Lowest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-small

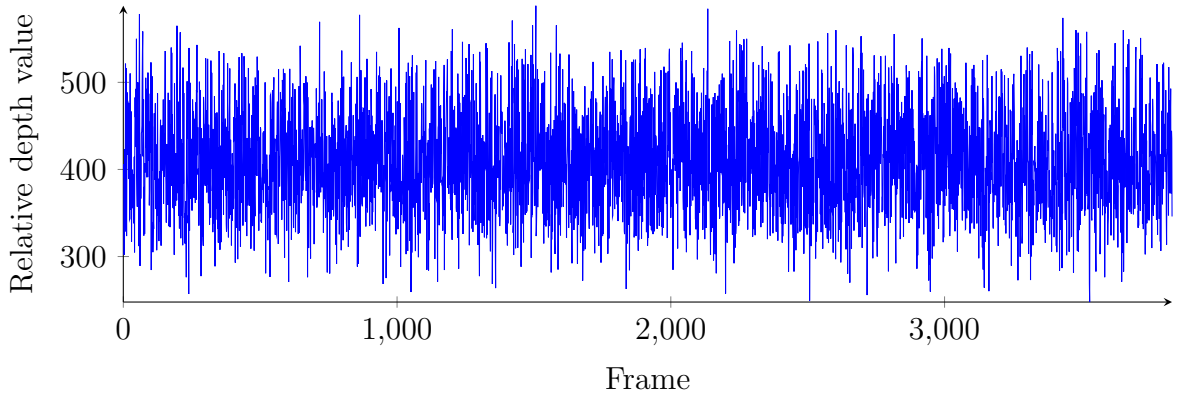

Figure 41. Highest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-small

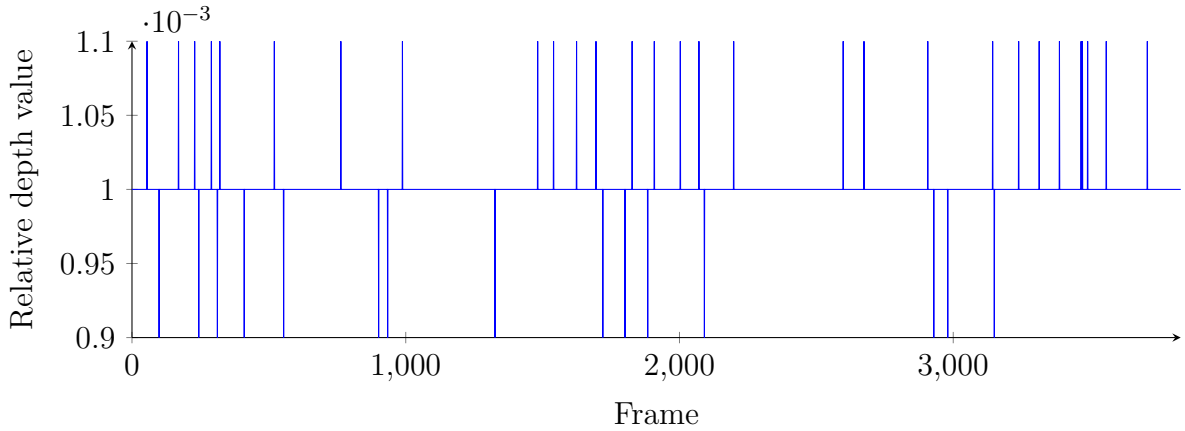

Figure 42. Lowest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-hybrid

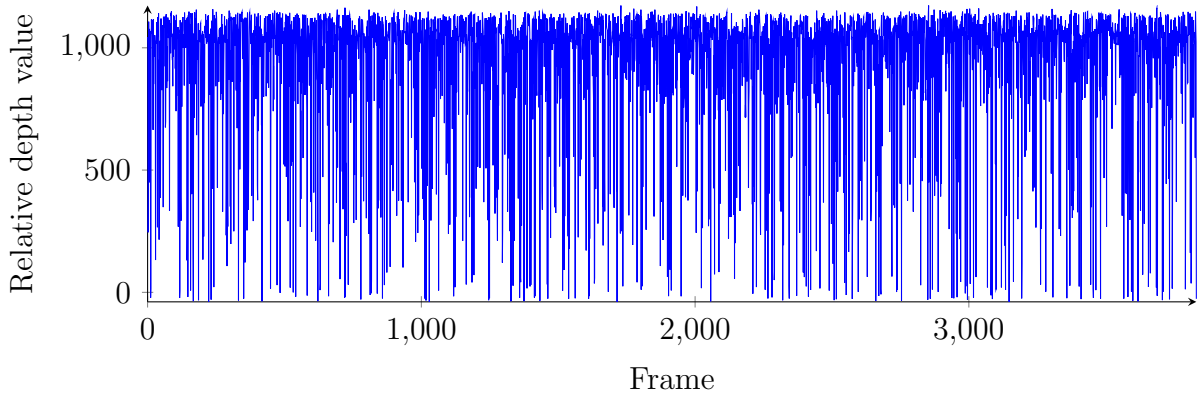

Figure 43. Highest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-hybrid

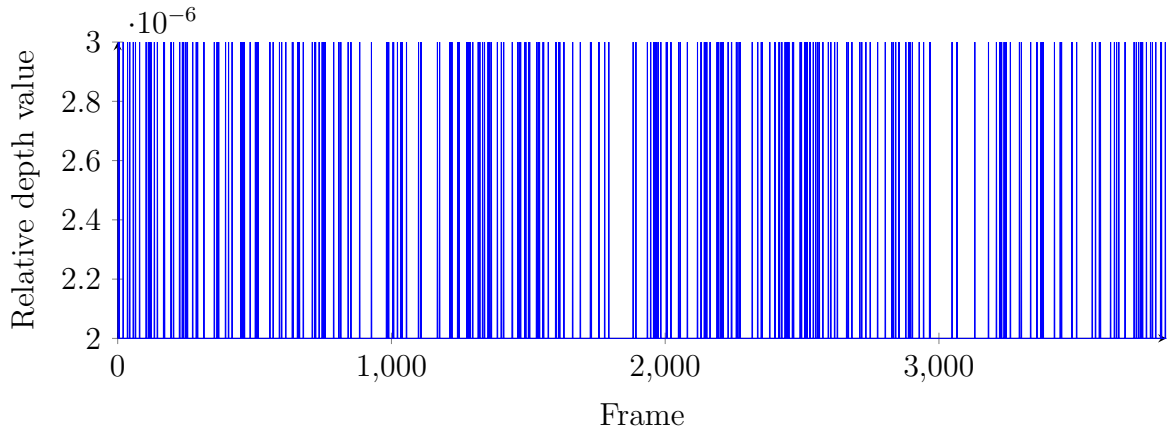

Figure 44. Lowest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-large

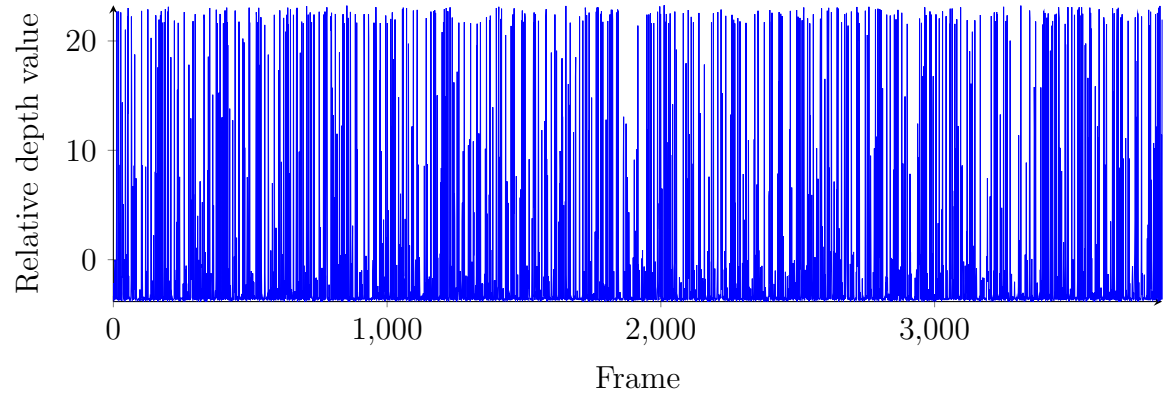

Figure 45. Highest variation of relative depth values.
Video used: Video 6 with tripod, Model used: MiDaS-large

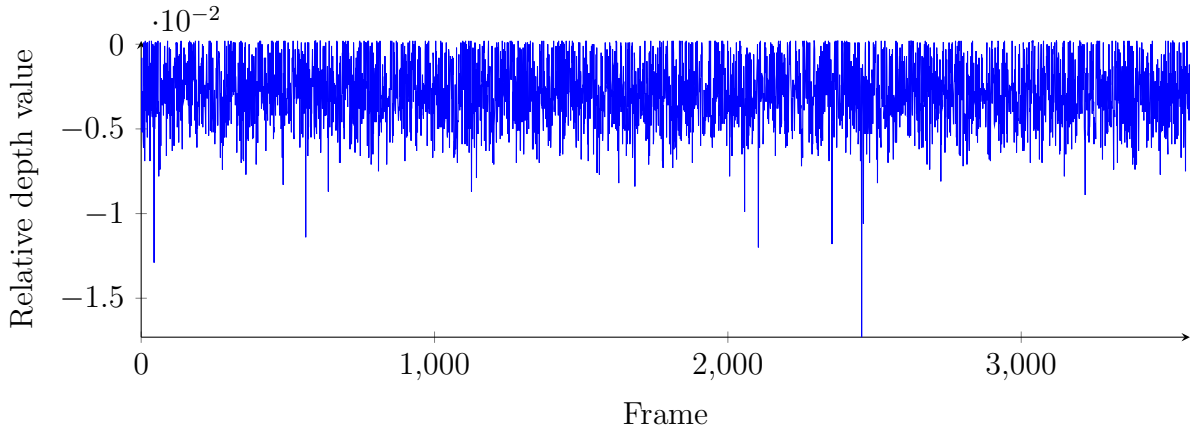

Figure 46. Lowest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-small

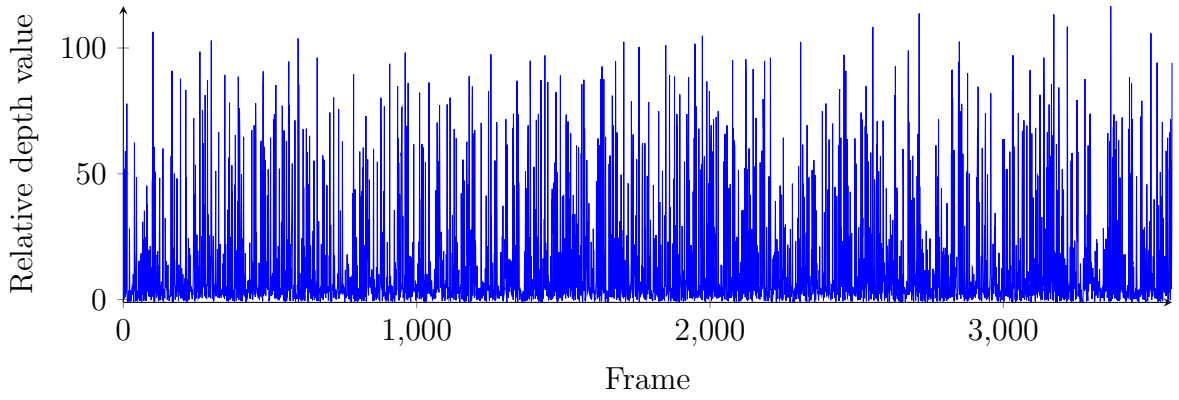

Figure 47. Highest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-small

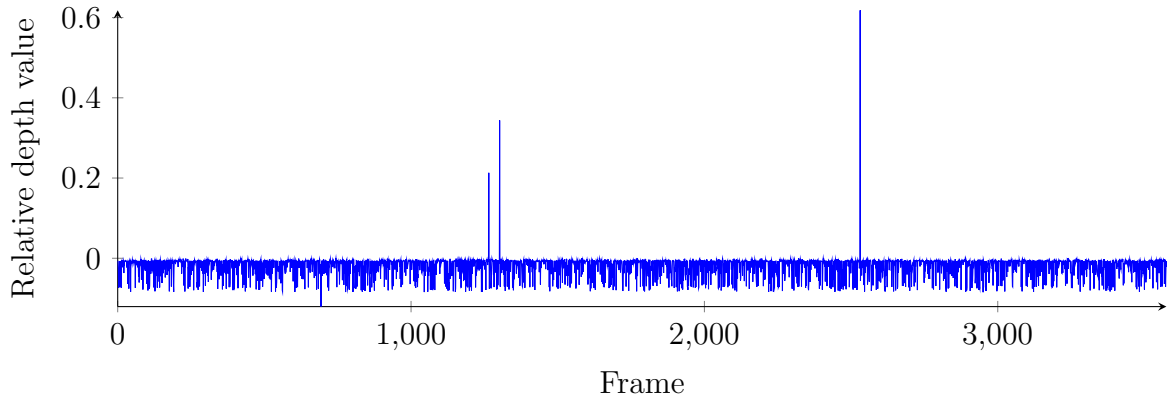

Figure 48. Lowest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-hybrid

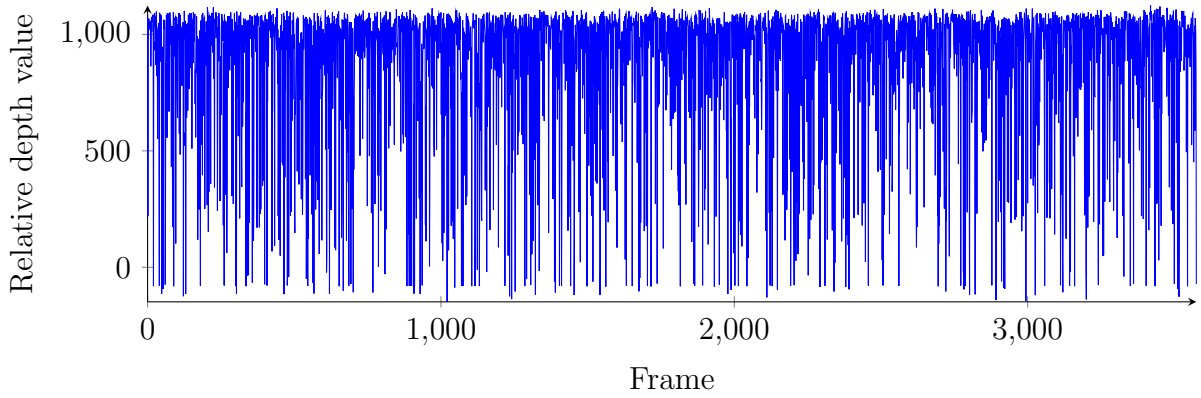

Figure 49. Highest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-hybrid

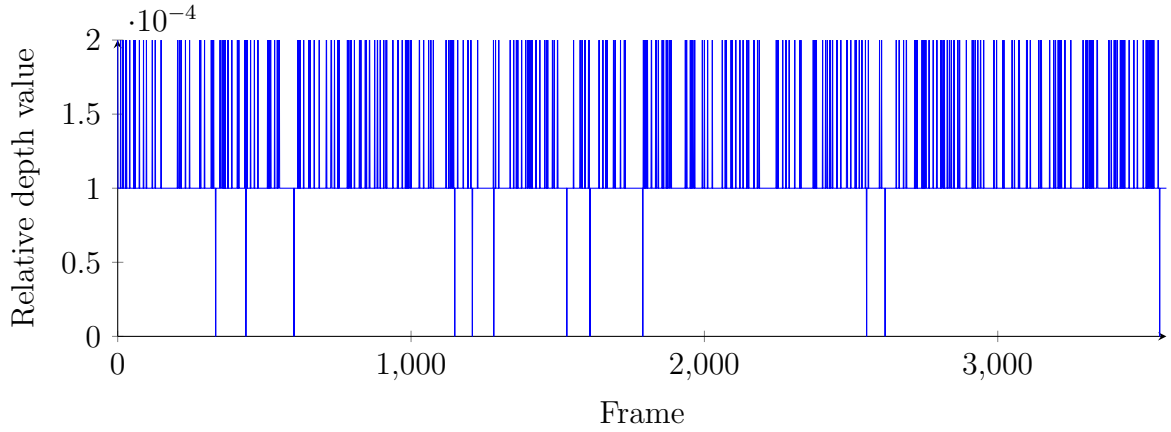

Figure 50. Lowest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-large

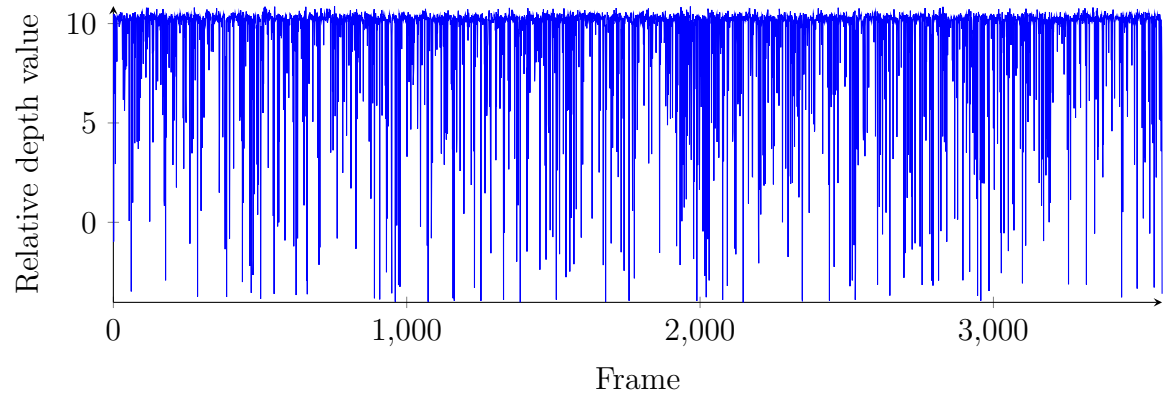

Figure 51. Highest variation of relative depth values.
Video used: Video 7 with tripod, Model used: MiDaS-large

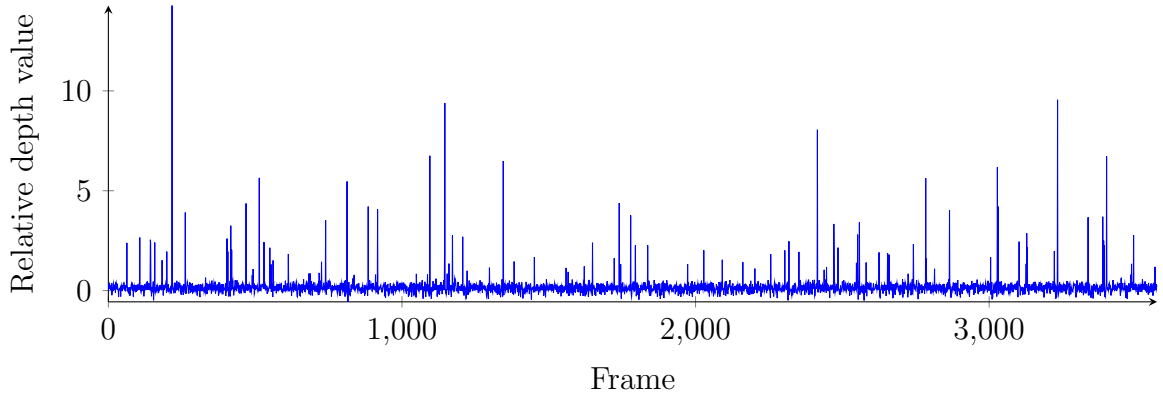

Figure 52. Lowest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-small

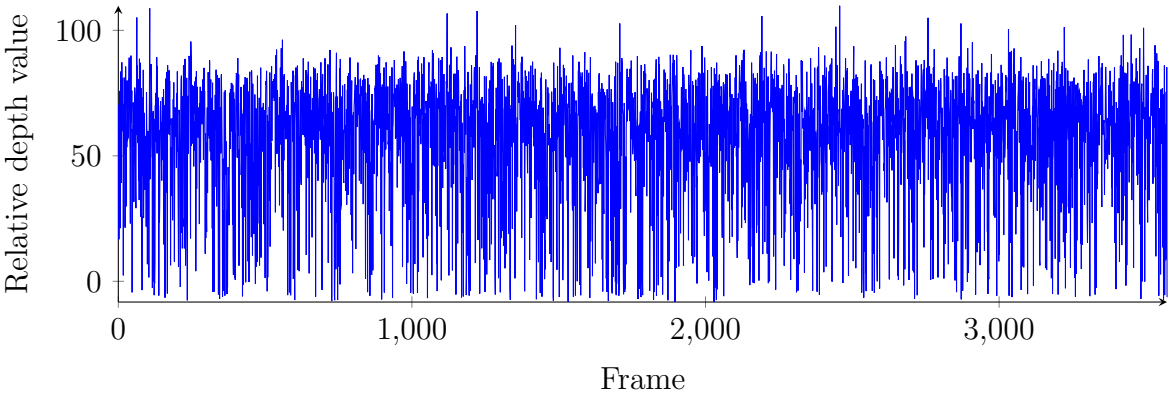

Figure 53. Highest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-small

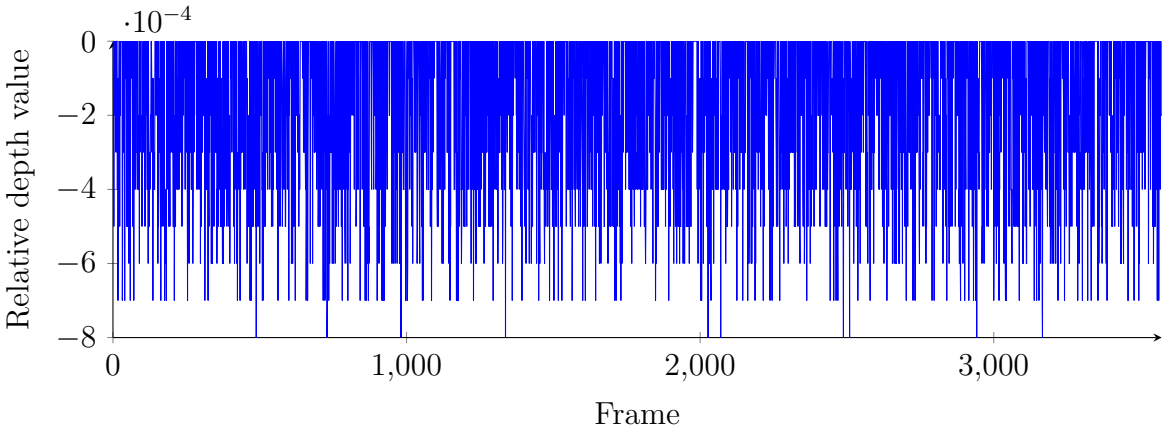

Figure 54. Lowest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-hybrid

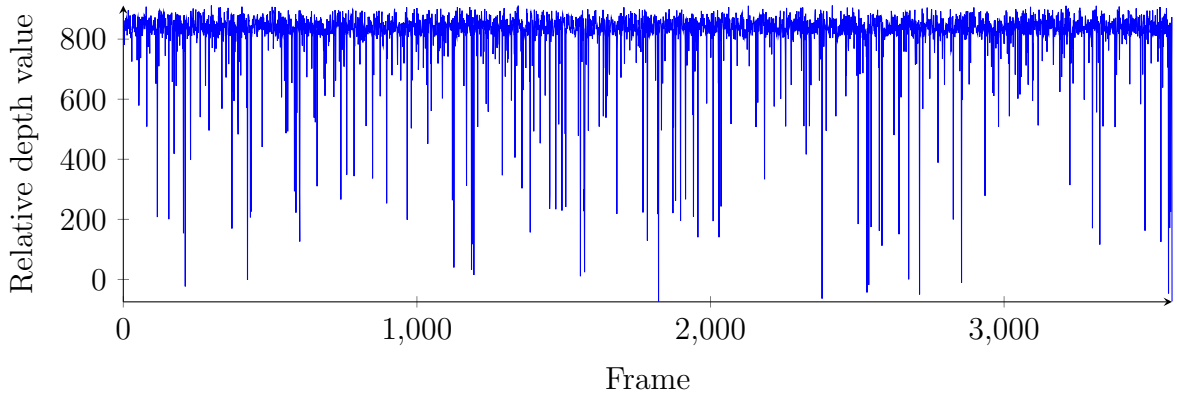

Figure 55. Highest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-hybrid

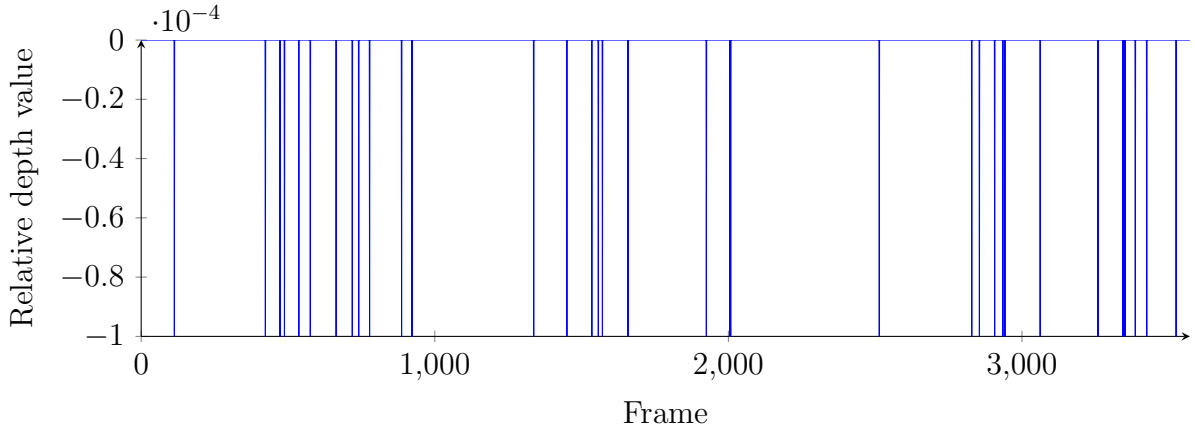

Figure 56. Lowest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-large

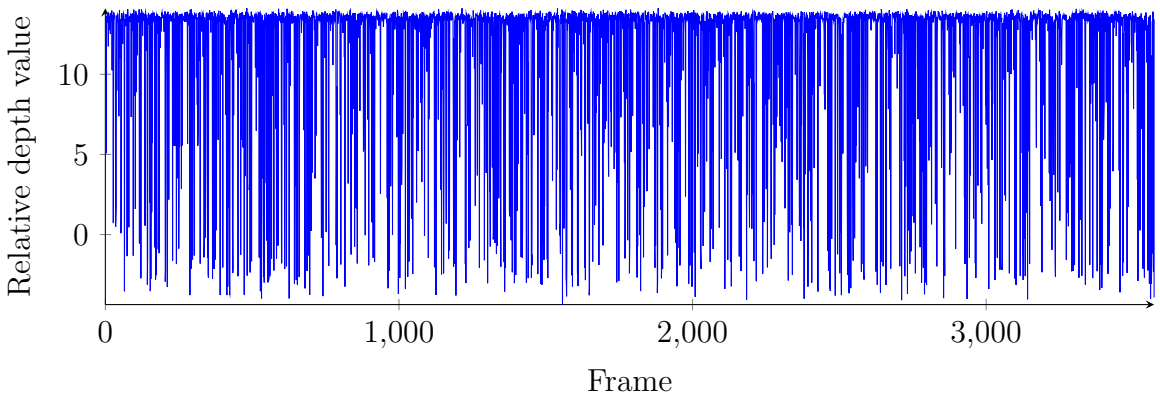

Figure 57. Highest variation of relative depth values.
Video used: Video 8 with tripod, Model used: MiDaS-large

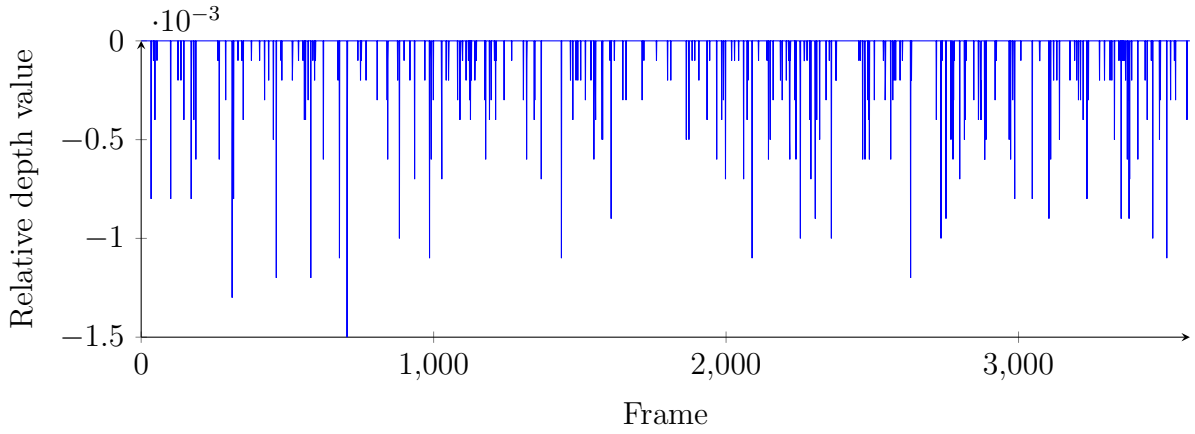

Figure 58. Lowest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-small

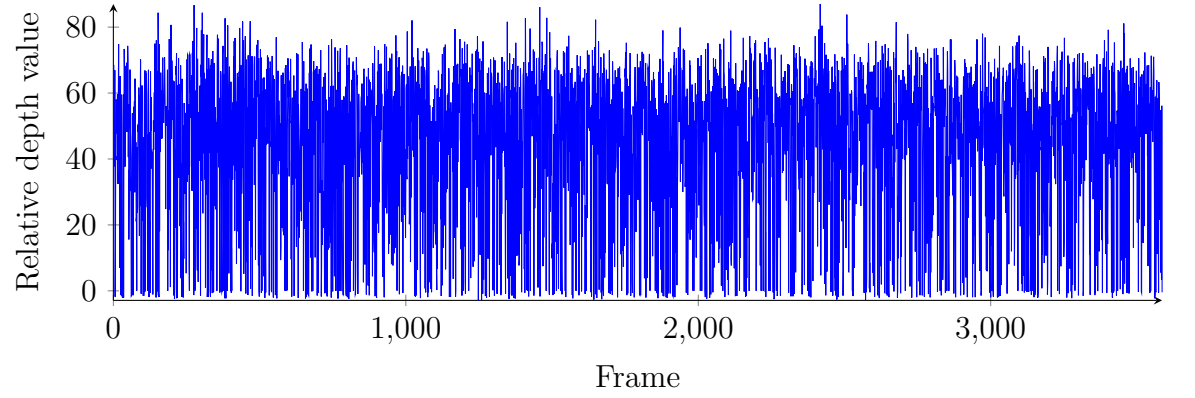

Figure 59. Highest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-small

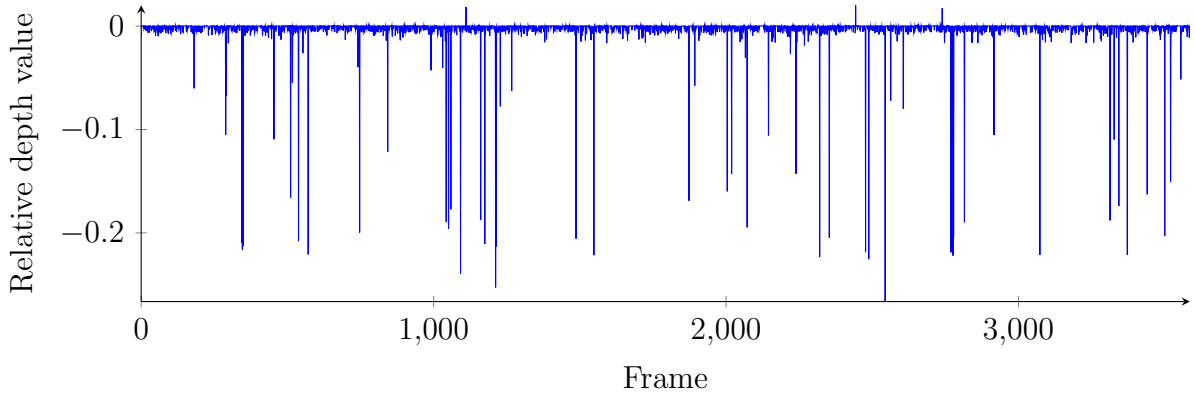

Figure 60. Lowest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-hybrid

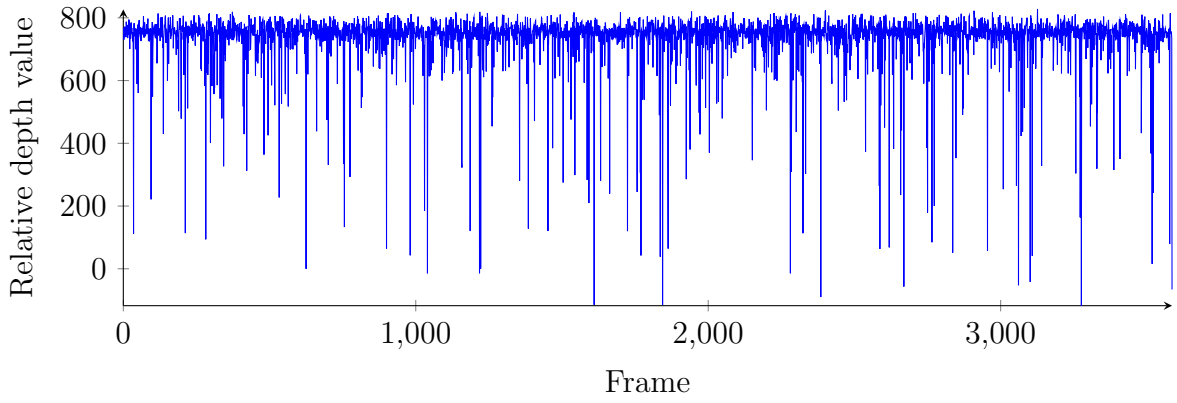

Figure 61. Highest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-hybrid

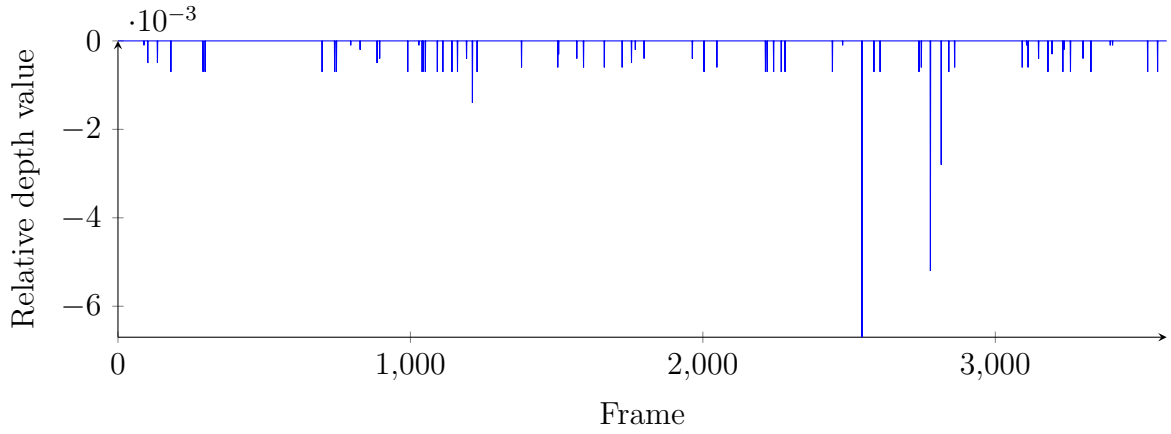

Figure 62. Lowest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-large

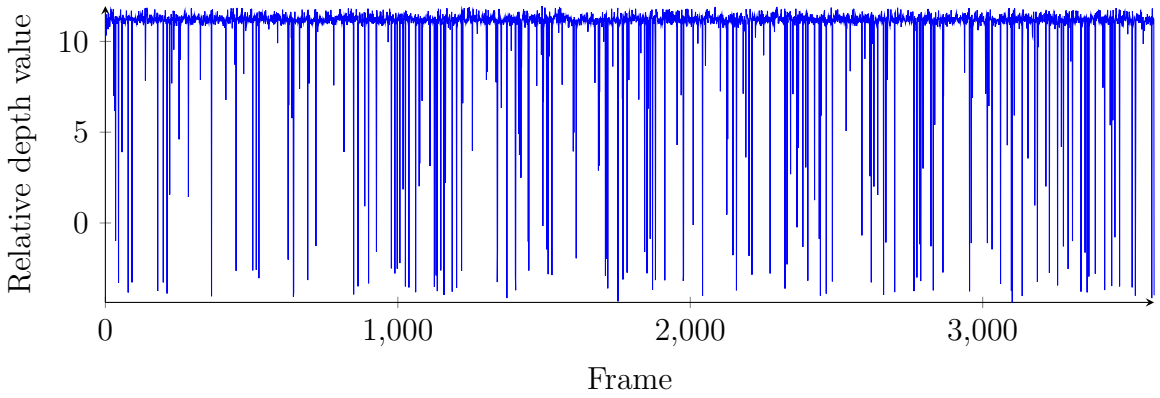

Figure 63. Highest variation of relative depth values.
Video used: Video 9 with tripod, Model used: MiDaS-large

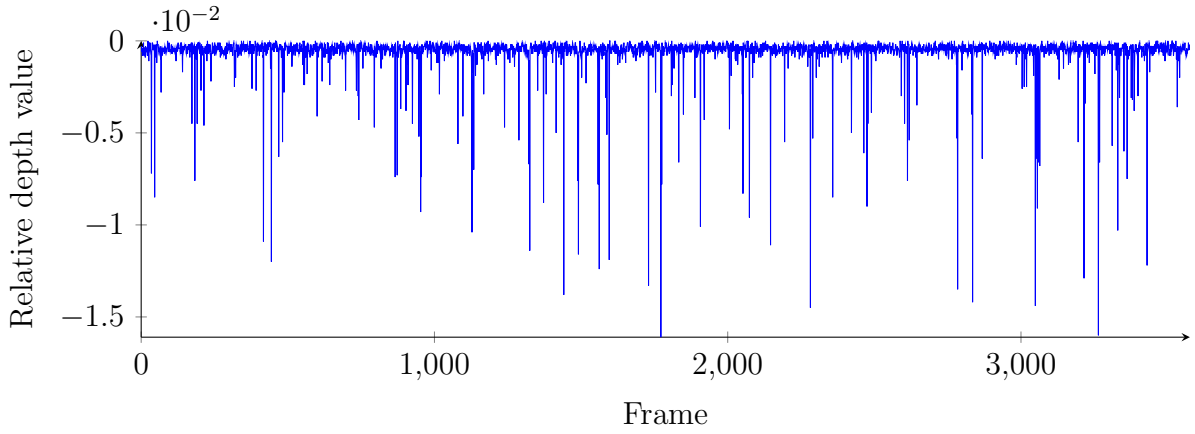

Figure 64. Lowest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-small

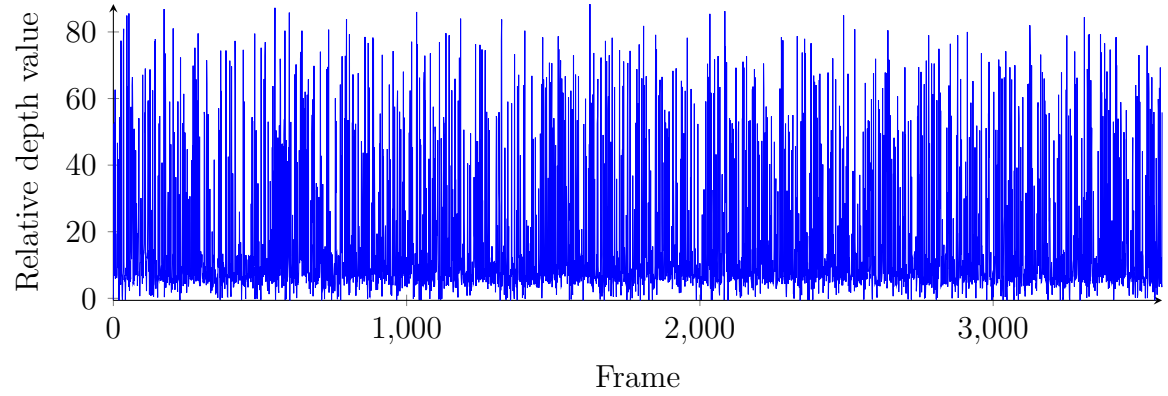

Figure 65. Highest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-small

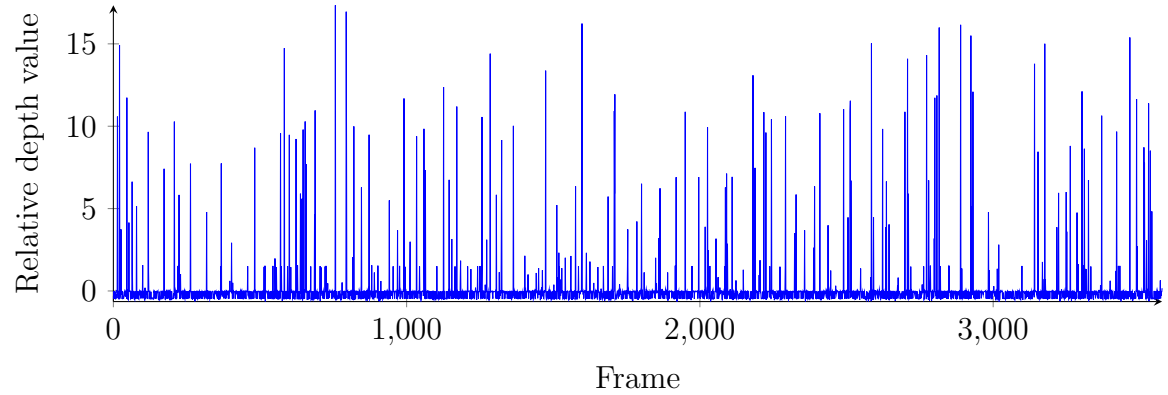

Figure 66. Lowest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-hybrid

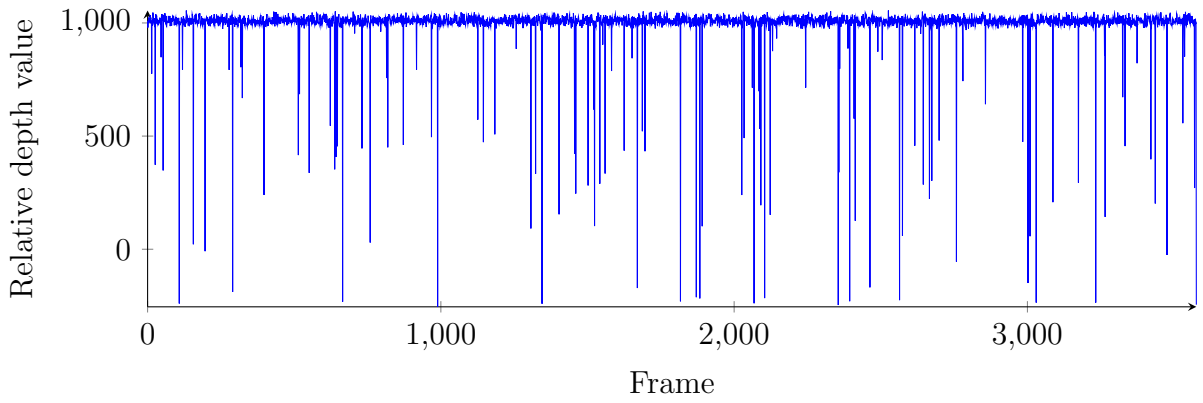

Figure 67. Highest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-hybrid

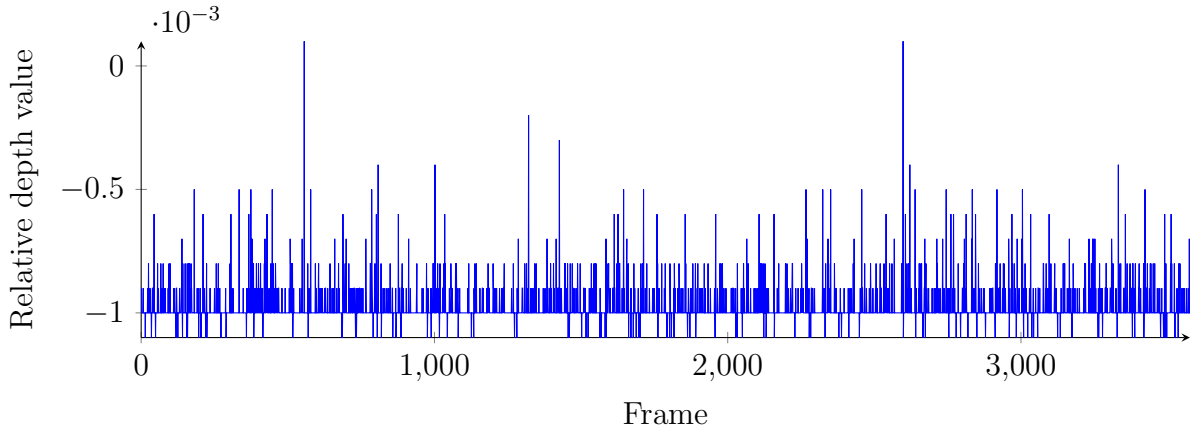

Figure 68. Lowest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-large

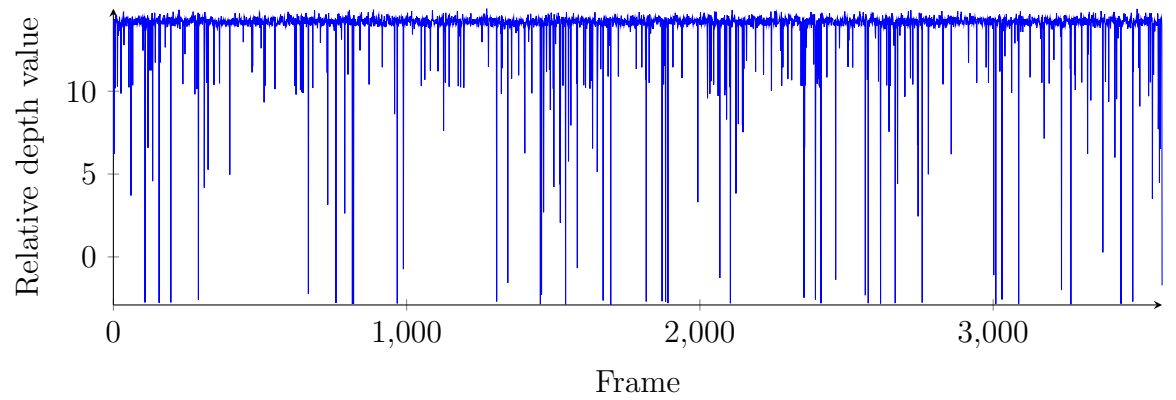

Figure 69. Highest variation of relative depth values.
Video used: Video 10 with tripod, Model used: MiDaS-large