Investigating Feature-Based Stereo Depth Estimation in Vehicular Contexts

---

Research Question:
How do the SuperPoint and ORB feature-extracting algorithms compare in terms of accuracy and computational efficiency in stereo block-matching based depth estimation?

Computer Science Extended Essay

Word Count: 3996

Table of Contents

1 Introduction

The main focus of this essay is to investigate the performance differences, in regard to accuracy and runtime, between the ORB (Orientated FAST and Rotated BRIEF) and SuperPoint feature extractors in a stereo depth estimation task. Stereo depth estimation refers to a process that involves determining the distance of objects in a scene, relative to the camera. This is done by analysing the difference in position between objects in the left and right images of a stereo camera system in a 'correspondence' process (Saxena, 2007). The underlying assumption of the process responsible is that both input images are undistorted or 'rectified' (Fahmy, 2013). The feature-extraction algorithm is pivotal in ensuring accurate results as it can facilitate the 'rectification' of the images for the correspondence process. These algorithms are used to identify and describe key, corresponding points in both the left and right images of the stereo system to help align the images themselves (Kitani, 2018).

In the past decade, the re-emergence of deep-learning research has induced a paradigm-shift in the 'world of knowledge' regarding computer vision. This has led to an increase in perception capabilities for autonomous / semi-autonomous vehicles, with manufacturers, like Tesla, using deep-learning algorithms alongside physical LiDAR sensors for depth estimation tasks amongst others (Karpathy, 2019). The use of deep learning-based algorithms for feature-extraction in particular, like SuperPoint, have become increasingly common. However, traditional feature-extraction algorithms are still implemented in various computer-vision applications, such as ORB and the ORB-SLAM algorithm (Simultaneous Localisation and Mapping) (Mur-Artal & Tardós, 2017). The dichotomy that has been created between emerging deep-learning feature-extraction and that of traditional techniques lead to the formulation of the research question: **How do the SuperPoint and ORB feature-extracting algorithms compare in terms of accuracy and computational efficiency in stereo block-matching based depth estimation?**

As mentioned previously with ORB-SLAM, the information computed from depth estimation algorithms can be integral for autonomous / semi-autonomous vehicular applications. Given the real-time computational, and accuracy needs for safe vehicular navigation, an experiment evaluating the capabilities of a traditional feature-extractor in comparison to that of a deep-learning based feature-extractor could prove especially useful in justifying the use of additional deep-learning applications regarding computer vision.

2 Background Information

2.1 Depth Estimation

In a stereo-camera setup, two cameras are positioned side by side with a fixed baseline (distance) between them, with each camera capturing a slightly different view. From this stereo-camera setup, a model can be derived in which a mathematical framework is produced where the depth of objects in a scene can be defined. The framework itself assumes that both cameras are fixed in place, and share the same intrinsic parameters which include the cameras' focal length and the principal point (Li, 2014). The diagram for this model can be seen in ***Figure 1***:

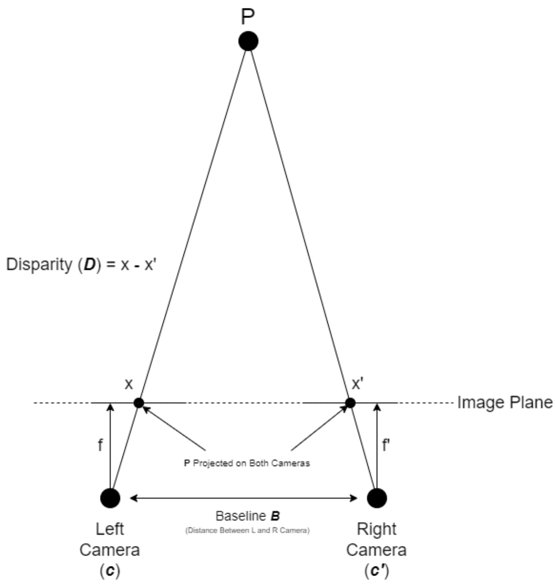


Figure 1: Diagram of the stereo camera model with a point P being projected on both cameras.

(Created by Author, 2023)

Regarding the focal length $\boldsymbol{f}$, it refers to the distance between the lens and the image sensor, which determines the field of view of the camera and the magnification of the image itself (Tsokos, 2014). The principal point $\boldsymbol{p}$ is the point on the image plane where the principal axis of the image sensor perpendicularly intersects with the image plane, the two-dimensional plane from which the images from the camera are projected (Alturki, 2017). An example of a model showing the focal length, principal point, and the principal axis for a singular camera can be seen in ***Figure 2:***

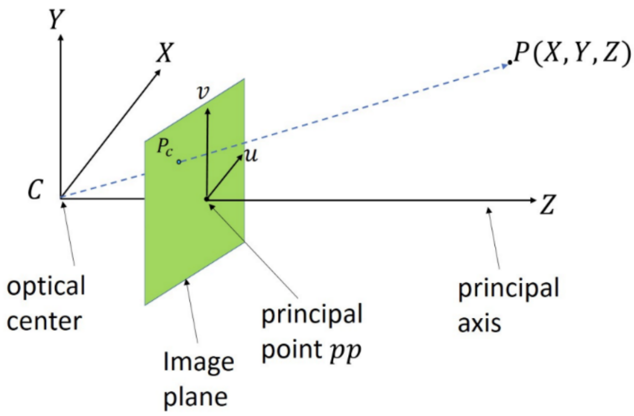


Figure 2: Diagram of a singular optical centre (camera sensor) with its principal point, and axis annotated along with a projected P.
(Alturki, 2017)

One of the methods of estimating depth from stereo cameras is through the use of disparity maps, images that provide information about the relative shift in the position of an object between the two cameras. To achieve this, one must find corresponding pixels in the images captured by the two cameras and compute the horizontal displacement between the pixels themselves. This horizontal displacement is known as the **disparity** and can be mathematically defined through the following equation:

$$Disparity = x - x'$$

This essentially means that with an object of a greater distance, the shift in its position between both cameras will be less pronounced than an object of a lesser distance. Thus, the relative distance of an object from the camera, or depth, can be determined with such disparity values. With all of these disparities mapped, it produces ***Figure 3b***, where points with a greater disparity are shown with a subsequently greater intensity of colour in the disparity map.

(a) **(b)**

Figure 3: a) Original image from the ‘KITTI’ dataset (KITTI, 2011) **b)** Computed disparity map from the given input image using the MiDaS network (Ranftl et al., 2022)

2.1.1 Stereo Block-Matching (SBM) Algorithm

In order to systematically determine the disparity for the whole image to produce these disparity maps, the Stereo Block-Matching (SBM) algorithm can be used. In essence, this algorithm determines disparity between two images from a stereo system by segmenting the left image into overlapping size-defined ‘blocks’, and using a search window that slides across a ‘scan-line’ (common y-coordinate) to find a matching block in the right image (Kitani, 2020). This approach to calculating disparity can be seen in ***Figure 4***:

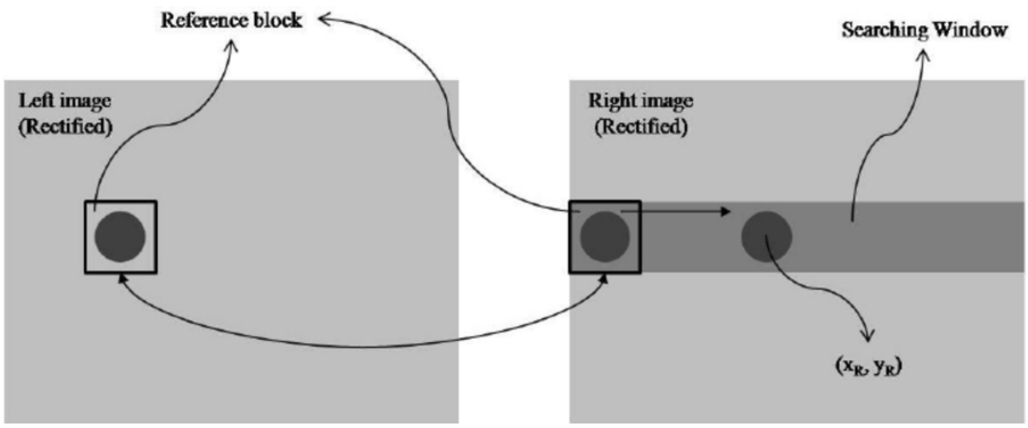


Figure 4: Stereo block-matching process for rectified images in diagrammatic form. (Chiang et al., 2011)

In order to compute this comparison, this block-matching algorithm uses a cost-function based similarity measure in the form of the Sum of Squared Differences (SSD) function (Kitani, 2020). This SSD function aims to quantify the dissimilarity between the block in the left image and a 'candidate' block in the right image by summing the squared differences of corresponding pixel intensities. This SSD function is defined as follows:

$$SSD(x, y) = \sum_{(x,y) \in I} \left(I_L(x, y) - I_R(x + d, y)\right)^2$$

(Kitani, 2020)

Where:

- $\boldsymbol{I_L}$ and $\boldsymbol{I_R}$ represent the image inputs provided by the left and right cameras respectively.
- $\boldsymbol{x}$ and $\boldsymbol{y}$ represent the pixel-coordinates being searched for the given images.
- $\boldsymbol{d}$ represents the value of disparity.

In order to find the appropriate disparity value to represent the block, the function finds the disparity value d in the function for the right image which minimizes the SSD function, which in turn, is also the best match for the image itself.

However, as mentioned previously, the main assumption for this algorithm is that both images are aligned across a common image plane or 'rectified'. In order to achieve this state of rectification for both images, the features in both images must be defined.

2.2 Feature-Extraction

The process of extracting features from an image serves as a fundamental sub-task for many applications in computer vision and image processing. The process itself involves pinpointing key-points within an image and describing them with a vector representation in the form of a 'descriptor' (MathWorks, n.d.). In the context of stereo depth-estimation, these meaningful points outlined by the feature-extraction algorithm are especially poignant during the ‘image rectification’ stage of the pipeline.

2.2.1 Image Rectification

Image rectification refers to the process of using matching ‘key-points’ within the left and right images of the stereo camera system to facilitate the geometric transformation (scaling and shearing) of the images themself. This transformation aligns both images in such a manner that the corresponding key-points of each image lie on a common image plane, such that the y-coordinate for point $\boldsymbol{p}$ in both cameras is the same (Kitani, 2020).

Typically, this is achieved through an estimation of the ‘fundamental matrix’ F, which encapsulates the relative orientation of the left camera to that of the right camera (Stachniss, 2020). Given the fundamental matrix, the transformations required to compute the rectification can occur. In order to compute the fundamental matrix, the position of the key-points are described using a system of coordinates called ‘homogenous coordinates’. This adds a common 3rd dimension of value 1 to the standard (x, y) vector to allow for the appropriate geometric transformations from the 3x3 ‘fundamental matrix’ F (Stachniss, 2020). An example of homogenous coordinates can be seen in the following with a key-point from both the left kp_L, and right camera kp_R.

$$kp_L = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \qquad kp_R = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Regarding the fundamental matrix F, this is represented through the following linear system.

$$(kp_L)^T F (kp_R) = 0$$

(Hartley, 2004)

Where:

- $\boldsymbol{kp_L}$ and $\boldsymbol{kp_R}$ denote the homogenous coordinates of their respective points on the left and right cameras.
- $\boldsymbol{F}$ represents the 3x3 fundamental matrix.

In order to compute the contents of the fundamental matrix from a set of matched key-points, the 'Eight Point' algorithm can then be used which uses 8 key-points to solve the homogenous linear system previously defined above (Stachniss, 2021).

After this transformation has occurred, the 'scanline' for the block-matching algorithm will therefore be at the same positions in both images to produce a more accurate computation of disparity values. An example of an image being rectified can be seen in the following figure:

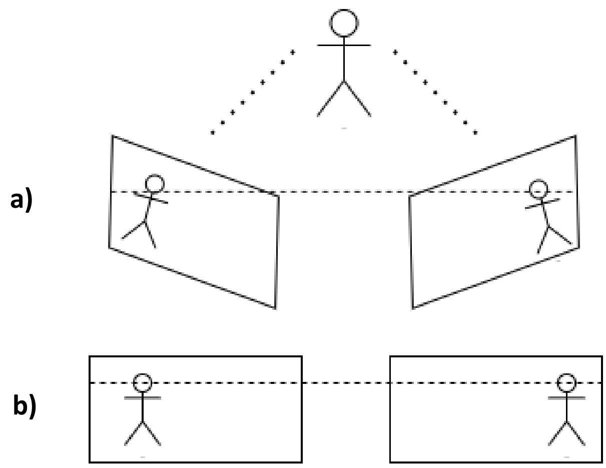
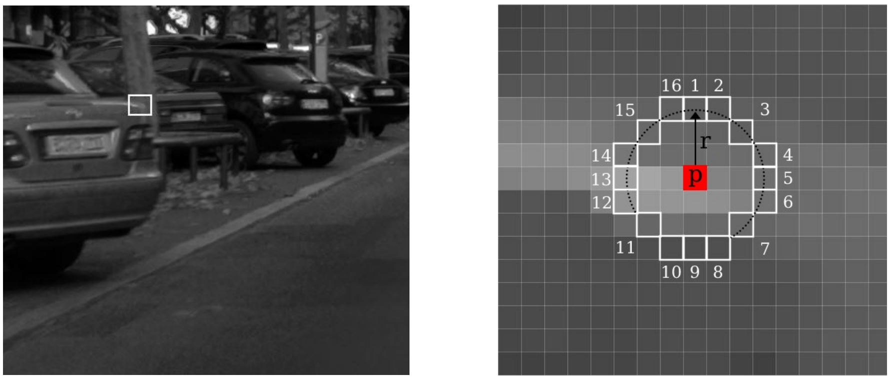


Figure 5: a) Original image captured by two cameras. **b)** Image after being rectified across a common image plane. (Created by Author, 2023)

2.2.2 Orientated FAST and Rotated Brief (ORB)

The Orientated FAST and Rotated Brief (ORB) feature-extracting algorithm, as outlined in the name, combines the FAST (Features from Accelerated Segment Test) 'key-point' detector, and the BRIEF (Binary Robust Independent Elementary Features) 'key-point' descriptor (Rublee et al., 2011).

The FAST algorithm functions as a ‘key-point’ detector through examining individual pixels and comparing the intensity value of the selected pixel against the surrounding pixels. The algorithm itself uses a circular search pattern with radius r, where n pixels are selected for comparison, and determines if the selected pixel can be classified as a ‘key-point’ by seeing if a contiguous amount of pixels in that selection window possess a lower or higher grayscale colour value (0 - 255) than the selected pixel (Rosten et al., 2010). This process can be seen in ***Figure 5a*** and ***Figure 5b***:



(a) **(b)**

Figure 5: a) The original input image with the pixel search window highlighted by a white box. (Cityscapes, 2020) **b)** A selected pixel ***p**,* being compared in terms of intensity against the neighbouring 16 pixels. (Created by Author, 2023)

After the key-points within the image have been identified with FAST, the BRIEF descriptor outputs a representation of the intensity pattern of the pixels surrounding the key-points that have been detected (Calonder et al., 2010). This representation comes in the form of a binary string, which is ideal for fast feature-matching between the left and right images in the stereo system as features in both images will have near-identical binary strings representing the key-point. Additionally, by providing a binary string descriptor to each key-point, the contiguous intensity pattern required means that if the image were to rotate, the same key-points can be described in the same manner with BRIEF (Calonder et al., 2010). An example of key-points extracted by the ORB algorithm in a given input image can be seen in ***Figure 6***:

Figure 6: Image from 'Cityscapes' dataset with 1000 key-points detected in the left camera of the stereo camera system by the ORB algorithm. (Created by Author, 2023)

With the features extracted in both left and right images, they can then be matched together as outlined previously with feature descriptors in addition to a 'brute-force' feature-matching algorithm. This algorithm searches for the best match between the feature-descriptors in both images to pair and then correlate the corresponding key-points (O'Reilley, n.d.). Matches, or correspondences, between the key-points in the left and right images can be seen in ***Figure 7:***

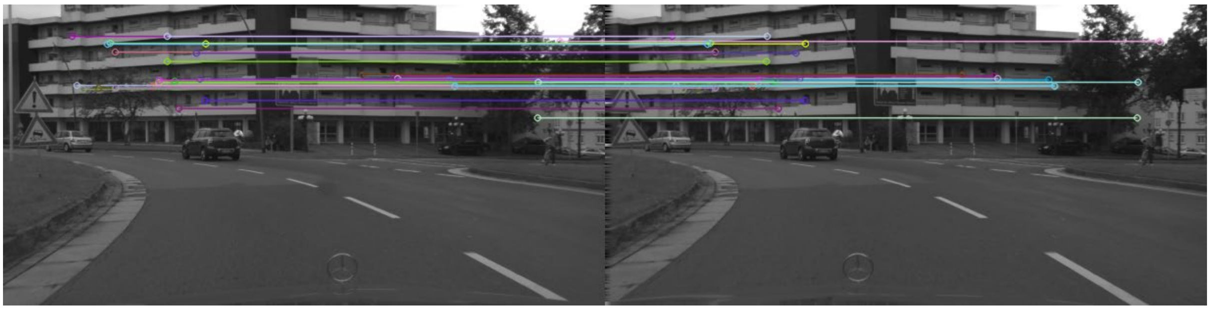


Figure 7: The best 10 ORB feature matches from the left and right cameras showcased with adjoining, coloured, lines. (Created by Author, 2023)

2.2.3 Self-Supervised Interest Point Detection and Description (SuperPoint)

Similar to ORB, SuperPoint also features a key-point detector and a descriptor. However, the main framework responsible for detecting key-points and descriptors in the image is that of a convolutional neural network (CNN), with both the key-point detector and descriptor being their own CNN (Detone et al., 2017). This general architecture for a stereo image pair can be seen in ***Figure 8:***

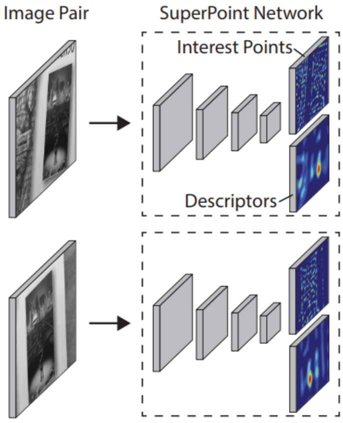


Figure 8: General architecture for the SuperPoint model. (Detone et al., 2017)

CNNs are a type of artificial neural network specifically designed for grid-structured data, such as those found in images with pixels. Its essence lies in using a multitude of filter layers, also known as kernels, that perform 'convolution' operations on the given input data through computations of the dot-product. This dot-product is made up of the kernel, sliding across the input data similar to the block-matching algorithm, and the data itself to extract a feature map of the image itself. Finally, a forward pass through the fully connected layers, where each neuron is connected to every previous and subsequent neuron, are used to output a prediction based on the features learned by the previous layers (IBM, n.d.). An example of a convolution operation to create a feature map can be seen in ***Figure 9***:

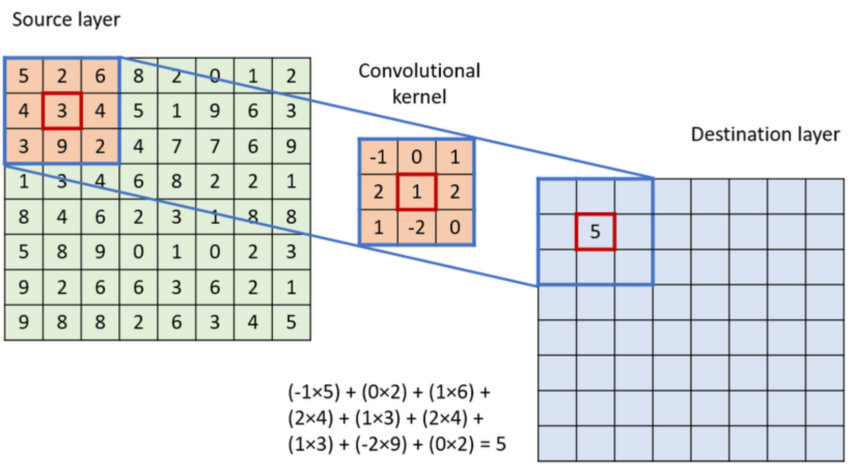


Figure 9: Diagram of a convolution operation with a 3x3 kernel on an 8x8 input image to an 8x8 destination layer.
(Podareanu et al., 2019)

In regard to training the key-point detector, a process called 'Homographic Adaptation' occurs, which makes the nature of the network itself self-supervised and also improves its ability to generalise to otherwise unfamiliar scenes that could occur to scale variance or rotations to the image (Detone et al., 2017). The general process for training the key-point detector using Homographic Adaptation can be seen in ***Figure 10***:

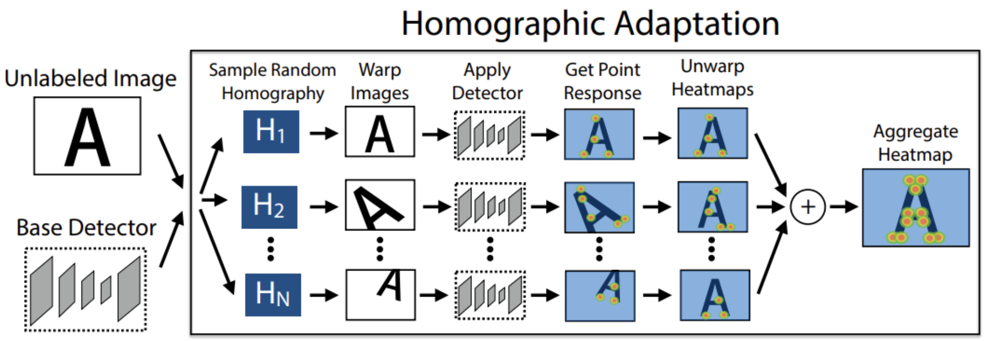


Figure 10: Self-supervised training of the interest-point detector in SuperPoint. (Detone et al., 2017)

Homographies refer to a mathematical transformation to an image plane at a given homogenous coordinate point. These transformations are used to describe image-to-image changes brought on by the movement of the cameras in terms of translations and rotations (Hartley, 2004).

$$\vec{P}' = H\vec{P}$$

Where:

- $\vec{P}'$ and $\vec{P}$ denote vectors with homogenous coordinates $[x \quad y \quad 1]^T$.
- H denotes the 3x3 matrix that transforms the $\vec{P}$ vector to get $\vec{P}'$

In Homographic Adaptation, warped duplicates of the input image are subjected to random homographies which are chosen at random to symbolise realistic camera shifts (Detone et al., 2017).

This process of applying this Homographic Adaptation technique also occurs iteratively throughout the training process, which progressively improves its ability to detect and extract repeatable key-points from images which will be resistant to changes in rotation, scale-changes, and occlusion, which is when an object is partially blocked by another object. This iterative process can be seen in the following figure:

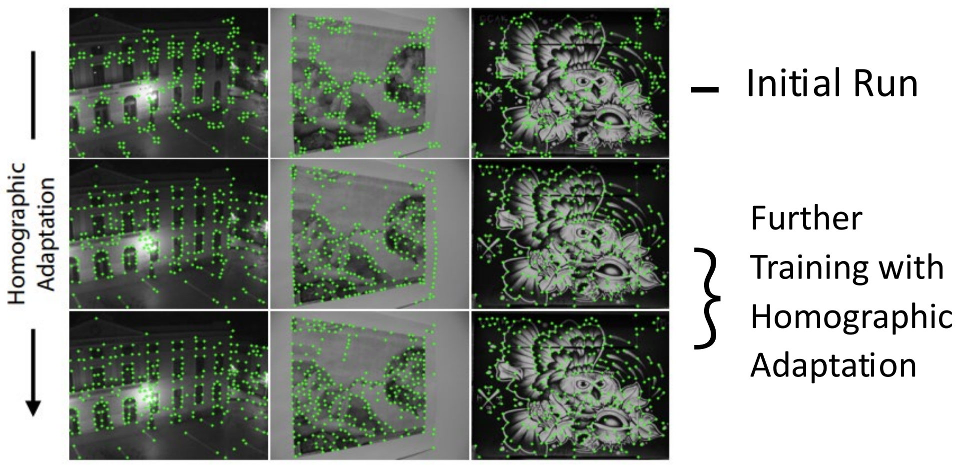


Figure 11: The SuperPoint key-point detector generalising to 3 images with self-supervised iterative training through Homographic Adaptation occurring.
(Detone et al., 2017)

In order to provide a descriptor for these key-points, fixed-size zones are first chosen around a key-point from the feature maps. This area is usually square and centred on the subject of interest with all of the activation values (output of a neuron) produced within the chosen region of feature map then being concatenated to produce vector-based representations of the feature for the descriptor. Essentially, this generated descriptor vector encodes the appearance and texture data near the interest point and describes the local picture structure (Detone et al., 2017).

3 Experimental Methodology

3.1 Dataset Used

As one of the main applications of depth estimation is with autonomous vehicles, the Cityscapes dataset will be used with the 'Berlin' testing dataset (Cityscapes, 2020) to explore the results in a vehicular context. This dataset also provides 'ground-truth' disparity maps for means of evaluating the computed disparity maps against (Cityscapes, 2020).

3.2 Experimental Variables

3.2.1 Independent Variables

The independent variable in this experiment will be the **number n of frames being computed for their disparity maps.** For each feature-extractor, I have chosen to have $n = 100$ frames as it should be enough to ascertain a trend in runtime performance, in addition to accuracy.

3.2.2 Dependent Variables

The following variables measuring accuracy in **Table 1** will be used:

Variable	Function Used	What it Measures
Mean Absolute Error **(MAE)**	`Sklearn.mean_absolute_error()`	MAE measures the average absolute difference between the intensity values in the disparity map, and their corresponding ground-truth values.
Root Mean Square Error **(RMSE)**	`Np.sqrt(mean_absolute_error)`	This metric provides a measure of the average magnitude of the errors between the computed and ground-truth disparity maps.

Structural Similarity Index Measure **(SSIM)**	`Skimage.ssim()`	Instead of looking for differences between the pixels of the ground truth and computed disparity maps, SSIM instead looks for similarities between the images and within the pixels themselves. For example, if the pixels in the images line up or have similar values. The value itself ranges from (0 to 1).

Table 1: Table of the metrics of accuracy used for this experiment.

Computational Efficiency:

To measure the computational efficiency of each algorithm, the **total runtime to compute disparity maps for the frames** will be recorded every 10 frames computed to showcase a trend and generalise a linear function representing the trend itself. To obtain the total runtime value, Python's `datetime.now()` function will be used.

3.2.3 Controlled Variables

To ensure that this experiment is free from any potential biases that might occur regarding hardware limitations and parameters, the following variables in ***Table 2*** will be controlled:

Variable	Description	Specifications
Hardware used to conduct the experiment.	The experiment will be carried out from my laptop.	**Processor:** Intel i5-1035G1 @ 1.00GHz, 1190 Mhz, 4 Core(s), 8 Logical Processor(s) **Memory:** 8GB 2133Mhz LPDDR3
Feature-matching algorithm used.	The same BF (Brute-Force) class from the OpenCV library will be used to match the features.	**Amount of Feature Matches Saved:** 30
Stereo correspondence algorithm used.	The same StereoBM class from the OpenCV library will be used for	**Search Window Size:** 16 Pixels

	correspondence and disparity computations.	**Possible Disparity Values per Pixel:** 11
The number of features extracted per frame analysed.	There will be 100 key-points from each frame will be detected and described.	N/A

Table 2: Table of the variables being controlled in the experiment.

3.3 Experimental Procedure

To collect the data for this experiment, the following procedure was undertaken:

1. Set up all of the classes (refer to ***Appendix A, B, C***) into an IDE, with the dataset of choice being present in the appropriate file paths of the `main` class.
2. Execute the code, setting the total amount of frames being analysed to the first 100 of the datasets.
3. Record the times and accuracy metrics that have been outputted to the terminal at every 10^{th} frame.
4. Repeat steps 2-3 in the procedure for 2 more trials.
5. Produce the average of the accuracy metrics and runtime totals from the 3 trials.
6. Change the feature-extractor being used and repeat steps 2-6.

4 Results

4.1 Tables of Results

After conducting the trials and computing the results *(**Appendix E**),* the average values have been collated and are shown in the tables 3-5 below:

ORB			
At nth Frame	**Avg. MAE**	**Avg. RMSE**	**Avg. SSIM**
10	17.41	31.47	0.55
20	43.89	47.59	0.46
30	47.51	49.19	0.47
40	41.25	47.48	0.48
50	59.48	63.78	0.34
60	35.23	38.65	0.54
70	45.24	48.21	0.51
80	61.39	64.87	0.39
90	64.23	70.27	0.37

100	41.95	44.07	0.49
Average	45.76	50.56	0.46

Table 3: Accuracy evaluation results for disparity maps computed with the ORB feature-extractor.

SuperPoint			
At nth Frame	**Avg. MAE**	**Avg. RMSE**	**Avg. SSIM**
10	25.02	32.82	0.59
20	42.20	41.31	0.48
30	45.74	43.21	0.49
40	39.70	43.34	0.51
50	31.23	44.65	0.53
60	40.91	37.54	0.51
70	47.12	42.93	0.48
80	48.86	49.51	0.46
90	43.20	47.24	0.47
100	43.73	48.87	0.42
Average	41.97	43.12	0.49

Table 4: Accuracy evaluation results for disparity maps computed with the SuperPoint feature-extractor.

	Average Time Taken to Compute Disparity Map (seconds)	
Frames Computed	**ORB**	**SuperPoint**
10	3.347138667	6.53280234
20	10.18629133	12.6337823
30	19.53433267	16.98984246
40	22.83107433	24.3616092
50	24.12630567	32.14613254
60	29.85934900	34.66010564
70	38.04704667	40.63889565
80	39.93260933	46.5628255
90	42.77988100	49.7469634
100	48.61496967	54.9393345

Table 5: The average time taken to compute disparity maps for each feature-extractor with a varying number n of frames computed.

4.2 Diagrammatic Representations of Data

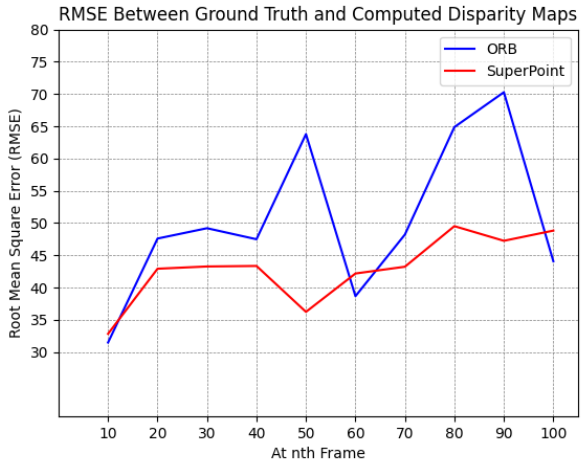


Figure 12: Root Mean Square Error (RMSE) graph between ORB and SuperPoint across 100 frames computed.

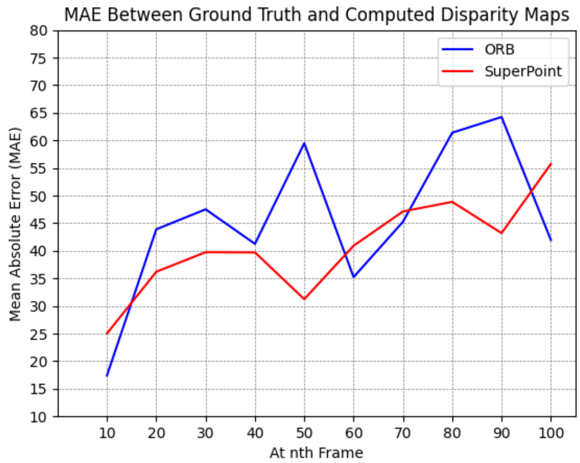


Figure 13: Mean Absolute Error (MAE) graph between ORB and SuperPoint across 100 frames computed.

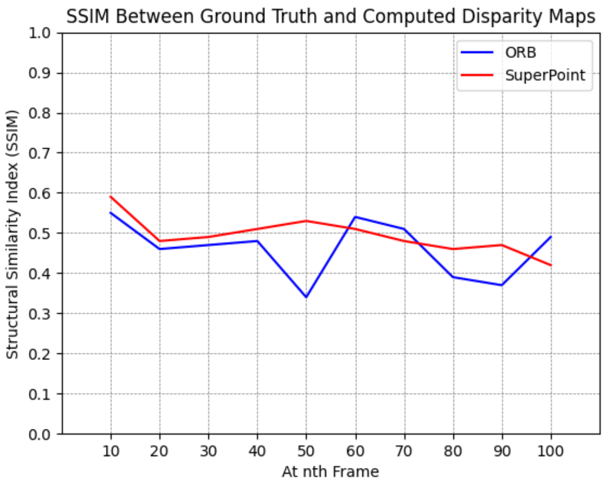


Figure 14: Structural Similarity Index (SSIM) comparison graph between ORB and SuperPoint across 100 frames computed.

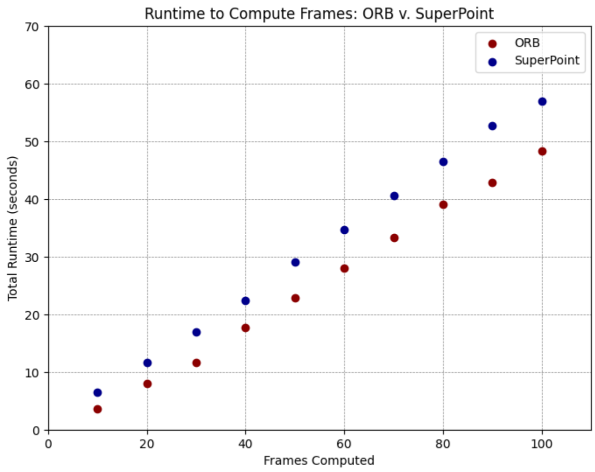


Figure 15: Graph of the total runtimes for both ORB and SuperPoint across the 100 frames computed.

4.3 Analysis of Results

Looking at the results of the experiment, it can be seen that SuperPoint's accuracy metrics were in fact better than ORBs. Expressed as a percentage, SuperPoint's average **RMSE of 43.15** is **14.66%** less than **ORB's** average at **50.56**. This means that SuperPoint's computed disparity values were, on average, closer to the ground-truth disparity maps with the average magnitude of errors being lower. Similarly, SuperPoint's average **MAE of 41.97** was **8.28%** less than **ORB's** average at 45.76. Meaning that, again, the disparity maps produced with SuperPoint-based rectification were more accurate. SuperPoint's greater accuracy can be seen again with the average **SSIM at 0.49** in contrast to ORBs at **0.46**, indicating a higher level of agreement in terms of patterns and pixel intensities in the image. Of course, this means that the rectification performed by SuperPoint was also more accurate. With these apparent limitations in ORB's accuracy, they are exemplified in particular with the computed disparity maps from the **80th** and **90th** frames with SSIMs of 0.39, and 0.37 respectively. This deviation can also be seen with the visible trough in ***Figure 14,*** and peaks in ***Figure 12*** and ***Figure 13*** at these two frames.

These deviations in the 80th and 90th frames' metrics can be attributed to ORB's inability to successfully generalise to scenarios with occlusions or challenging lighting conditions, which can be due to changes in texture patterns that may affect key-point matching and detection. These scenarios themselves are quite typically faced in on-road navigation in urban areas, which was evident in the image itself, with them occurring in low-light, reflections or glare. Additional scenarios include adverse weather conditions and occlusions that naturally occur in congested traffic scenarios. As a result, the performance of ORB in these situations could be compromised, leading to increased errors in the disparity maps due to the affected rectification process.

In addition to the higher degree of accuracy exhibited by SuperPoint's disparity maps, they also displayed a greater consistency with the results. This consistency can be seen through **the range of data** produced with **16.69 for RMSE**, **29.42 for MAE**, and **0.17 for SSIM** respectively. This is in stark contrast to **ORB's accuracy spread** of **RMSE** which was **31.8% greater** than SuperPoint at **38.8**, **MAE** which was also greater by an even greater factor of **52.4%** at **46.82**, and the SSIM which was only **21%** greater in its spread at **0.21**. This consistency implies that SuperPoint experienced fewer errors in the rectification process, resulting in more reliable and visually accurate estimations of depth. In contrast, ORB exhibited quite volatile variations in its accuracy metrics, particularly when computing **the 80th and 90th frame** as discussed previously.

However, with the runtime, it can be seen in **Figure 15** that SuperPoint exhibited a higher computational cost compared to ORB with a **~7 second greater** runtime for the computation of the disparity maps. This means that the system running the experiment experienced a greater load, therefore being less computationally efficient in comparison to the ORB feature-extractor. In regard to extrapolating a runtime result to highlight this difference, the results shown in ***Table 5*** and ***Figure 15*** show that the general shape of trend is that of a Linearithmic trend. This Linearithmic trend can be denoted through the function: $n \log n$ (Bonaci, 2018) and when fitted to the points shown in **Figure 15,** they produce the following functions for ORB and SuperPoint runtimes respectively:

ORB $OR(n)$:

$$OR(n) = 0.239n \log n + 1.752$$

SuperPoint $S(n)$:

$$S(n) = 0.272n \log n + 4.926$$

With these functions fitted to these points, it produces the following graph:

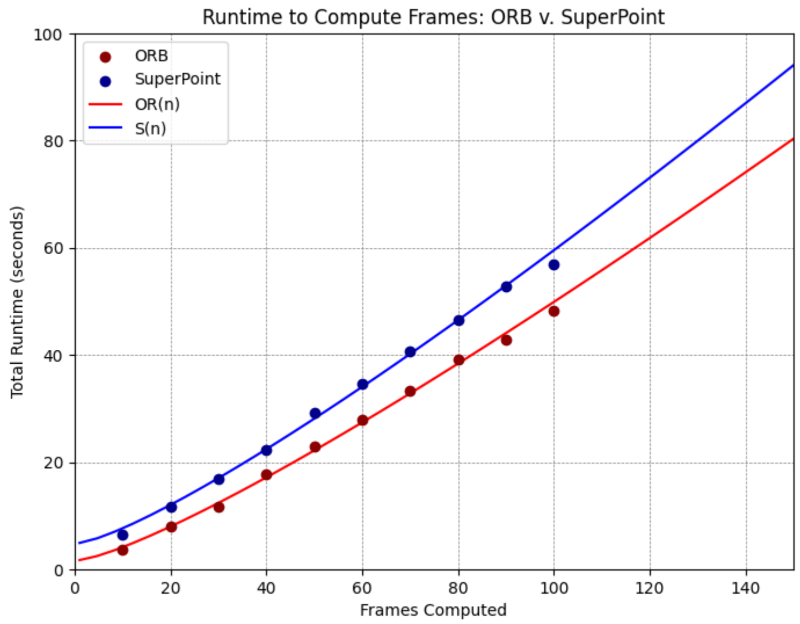


Figure 16: Graph of total runtime to compute disparity maps between ORB and SuperPoint with fitted Linearithmic functions $S(n)$ and $OR(n)$.

Firstly, to verify the Linearithmic relationship of these runtime functions and the data points on ***Figure 16**,* Pearson's correlation coefficient r can be used. After computing this coefficient, they show that there is a very strong Linearithmic relationship with r values of 0.9965 for $OR(n)$ and 0.9996 for $S(n)$ respectively. Although ***Figure 16*** already shows a large discrepancy in computational time, the rate at which each disparity map is computed for each feature extractor could further reinforce the extent of this discrepancy. This can be done through finding the first derivative of the functions, which produces as a result:

$$OR'(n) = \frac{239 \ln n + 239}{1000 \ln 10}$$

$$S'(n) = \frac{34 \ln n + 34}{125}$$

For the ORB function $OR'(n)$ at $n = 1000$ frames computed, this would result in $\mathbf{0.82}$ **seconds per disparity map computed**. For $S'(n)$ at n, this would result in $\mathbf{2.15}$ **seconds per disparity map computed**. According to these values, estimating depth with ORB-based rectification should provide greater performance by a factor of 1.33 seconds per disparity map, or a 61.86% increase in runtime in comparison to SuperPoint. This discrepancy between ORB and SuperPoint in their runtimes can be attributed to the inherent need for parallel computing in CNNs to perform computations simultaneously. As the system conducting the experiment only features a single multi-core processor, the workload could not be distributed adequately which lead to the slowed runtime for SuperPoint. Of course, this issue could be resolved with the use of a GPU that features parallel computing cores for graphics-based tasks, such as CUDA-based GPUs from Nvidia (Nvidia, 2017). However, this poses the issue of an additional hardware-based cost incurred in order to consistently harness a real-time capable deep-learning model - which is otherwise not required with ORB. Finally, due to the presence of a y-intercept for these functions for runtime, this means that there is a level of systematic error present in the experiment.

5 Evaluation and Limitations

One of the main limitations to this experiment is that the results of the accuracy metrics for both feature-extractors were very much bounded by the stereo block-matching algorithm used. This stereo block-matching algorithm produced quite mediocre disparity maps, meaning that it may not be fully representative of the feature-extractors ability to generalise to these images. Despite this limitation, there was still a clear difference in the results of the accuracy as well as the efficiency. This means a general conclusion could still be reached in terms of comparing the two algorithms. With respect to the systematic errors in regard to the runtime functions, they indicate that there is some level of systematic error due to the fact that at $n = 0$, the total runtime should be at 0. However, due to the presence of a y-intercept, this clearly is not the case. The systematic errors present in the trials can possibly be explained by a variation in resource allocation due to background processes with the CPU, and power fluctuations throughout the system which might have affected both algorithms' runtime performance.

5.1 Other Modes of Exploration

Other modes for exploration that might be interesting in regard to comparing the performance of ORB and SuperPoint might be:

- Dataset Variation: Utilize this depth estimation problem with a dataset that contains varying weather conditions that might be present in a vehicular scenario, providing conditions that may offer further occlusion-present situations.
- SLAM Implementation: See how well the depth information computed with the help of these feature-extractors can translate into mapping the environment as well as localising the camera that could be evaluated with rotational errors, and displacement errors.

6 Conclusion

This essay found how the SuperPoint and ORB feature-extracting algorithms compared in terms of accuracy and the computational efficiency in the computation of disparity maps for the purposes of a semantic understanding of depth in a given scene. From the results of the trials, depth estimation conducted with SuperPoint in the rectification process produced average accuracy metrics of 43.12 for RMSE, 41.97 for MAE, and 0.49 SSIM. In comparison, disparity maps computed with ORB rectification produced lesser metrics, accuracy-wise, at 50.56 RMSE, 45.76 MAE and 0.46 SSIM. Despite the greater accuracy from SuperPoint, there is an apparent trade off in regard to the computational efficiency of the algorithm itself. After computing the disparity maps for the testing dataset of 100 frames, the record total runtime was ~7 seconds greater than ORB at 54.54 seconds in comparison to 48.61 seconds. As a result, this implies that SuperPoint's capabilities are less suited for real-time applications without the sufficient hardware for parallel computations, hindering its potential use in smaller scale cases of navigation that require depth perception.

To answer the research question, **the use of the SuperPoint feature-extractor will produce more accurate computations of disparity maps with greater consistency. However, the ORB feature-extractor will be more computationally efficient in computing these disparity maps.**

	ORB	**SuperPoint**
Average MAE	45.76	41.97
Average RMSE	50.56	43.12
Average SSIM	0.46	0.49
Time to Compute 100 Frames (s)	48.61 seconds	54.54 seconds
Time Per Disparity Map Computed at 1000 Frames(s)	0.82 seconds per map	2.15 seconds per map

Table 6: Summary comparison table for ORB to SuperPoint.

Bibliography

Alturki, A. (2017). Principal Point Determination for Camera Calibration [Doctoral Thesis]. https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=dayton1500326474390507&disposition=inline

Bonaci, T., & Slaughter, A. (2018). CS5002: Discrete Structures: Introduction to Algorithms. Carnegie Mellon University. https://course.ccs.neu.edu/cs5002f18-seattle/lects/cs5002_lect9_fall18_notes.pdf

Calonder, M., Lepetit, V., Strecha, C., & Fua, P. (2010). BRIEF: Binary Robust Independent Elementary Features. https://www.cs.ubc.ca/~lowe/525/papers/calonder_eccv10.pdf

Chiang, M.-H., Lin, H.-T., & Hou, C.-L. (2011). Development of a Stereo Vision Measurement System for a 3D Three-Axial Pneumatic Parallel Mechanism Robot Arm. Sensors, 11(2), 2257–2281. https://doi.org/10.3390/s110202257

Cityscapes. (2020). Cityscapes Dataset – Semantic Understanding of Urban Street Scenes. Cityscapes. https://www.cityscapes-dataset.com/

Detone, D., Malisiewicz, T., & Rabinovich, A. (2017). SuperPoint: Self-Supervised Interest Point Detection and Description. arXiv. https://arxiv.org/pdf/1712.07629.pdf

Fahmy, A. A. (2013). Stereo Vision Based Depth Estimation Algorithm In Uncalibrated Rectification. Taif University. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=32e9d2bfe7d347662591cfb92c8309e62ea7fe51

Hartley, R., & Zisserman, A. (2004). Multiple View Geometry in Computer Vision. Cambridge University Press.

IBM. (n.d.). What are Convolutional Neural Networks? | IBM. Www.ibm.com. https://www.ibm.com/topics/convolutional-neural-networks

Karpathy, A. (2019). How AI Powers Self-Driving Tesla with Elon Musk and Andrej Karpathy. Www.youtube.com. https://www.youtube.com/watch?v=FnFksQo-yEY

Kitani, K. (n.d.). Stereo Vision 16-385 Computer Vision. Carnegie Mellon University. https://www.cs.cmu.edu/~16385/s17/Slides/13.1_Stereo_Rectification.pdf

Kitani, K. (2020). Stereo Matching 16-385 Computer Vision (Kris Kitani). Carnegie Mellon University. https://www.cs.cmu.edu/~16385/s17/Slides/13.2_Stereo_Matching.pdf

KITTI (Karlsruhe Institute of Technology). (2011). The KITTI Vision Benchmark Suite. Www.cvlibs.net. https://www.cvlibs.net/datasets/kitti/raw_data.php

Li, F.-F. (2014). Lectures 9&10: Stereo Vision. Vision.stanford.edu; Stanford Vision Lab. http://vision.stanford.edu/teaching/cs131_fall1415/lectures/lecture9_10_stereo_cs131.pdf

Magic Leap. (2020, February 15). SuperPoint Weights File and Demo Script. GitHub. https://github.com/magicleap/SuperPointPretrainedNetwork

MathWorks. (n.d.). Feature Extraction. Au.mathworks.com. https://au.mathworks.com/discovery/feature-extraction.html

Mur-Artal, R., & Tardos, J. D. (2017). ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. IEEE Transactions on Robotics, 33(5), 1255–1262. https://doi.org/10.1109/tro.2017.2705103

Nvidia. (2017). CUDA Zone - Library of Resources. NVIDIA Developer. https://developer.nvidia.com/cuda-zone#:~:text=CUDA%C2%AE%20is%20a%20parallel

O'Reilley. (n.d.). Brute-force matching - Learning OpenCV 4 Computer Vision with Python 3 [Book]. Www.oreilly.com. https://www.oreilly.com/library/view/learning-opencv-4/9781789531619/b838910f-5ceb-4181-a3dc-8d57aa21e6f0.xhtml#:~:text=A%20brute%2Dforce%20matcher%20is

Podareanu, D., Codreanu, V., Aigner, S., & Leeuwen, C. N. (2019). Best Practice Guide-Deep Learning. Www.semanticscholar.org. https://www.semanticscholar.org/paper/Best-Practice-Guide-Deep-Learning-Podareanu-Codreanu/6c00d026e1048ed27e4ff66e8a287baa8febdfbf

Ranftl, R., Lasinger, K., Hafner, D., Schindler, K., & Koltun, V. (2022). Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer. IEEE Transactions on Pattern Analysis and Machine Intelligence, 44(3), 1623–1637. https://doi.org/10.1109/tpami.2020.3019967

Rosten, E., Porter, R., & Drummond, T. (2010). Faster and Better: A Machine Learning Approach to Corner Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(1), 105–119. https://doi.org/10.1109/tpami.2008.275

Rublee, E., Rabaud, V., Konolige, K., & Bradski, G. (2011, November 1). ORB: An efficient alternative to SIFT or SURF. IEEE Xplore. https://doi.org/10.1109/ICCV.2011.6126544

Savnani, P. (2022, March 27). Depth Estimation Using Stereovision. GitHub. https://github.com/savnani5/Depth-Estimation-using-Stereovision

Saxena, A., Schulte, J., & Ng, A. (2007). Depth Estimation using Monocular and Stereo Cues. International Joint Conference on Artificial Intelligence, 7, 2197–2203. https://www.cs.cornell.edu/~asaxena/learningdepth/DepthEstimation_Saxena_IJCAI.pdf

Stachniss, C. (2020). Fundamental and Essential Matrix - 5 Minutes with Cyrill. Www.youtube.com. https://www.youtube.com/watch?v=auhpPoAqprk

Stachniss, C. (2021). 8 Point Algorithm - 5 Minutes with Cyrill. Www.youtube.com. https://www.youtube.com/watch?v=z92eUJjIJeY&t

Tsokos, K. A. (2014). Physics for the IB diploma (6th ed., pp. 125–126). Cambridge University Press. (Original work published 1998)

Appendix

Appendix A: Code for Depth Estimation (Savani, 2022) (https://github.com/savnani5/Depth-Estimation-using-Stereovision)

Main Class

```
import math

import cv2

import random as rd

import numpy as np

import matplotlib.pyplot as plt

import datetime


from calibration import draw_keypoints_and_match, drawlines, RANSAC_F_matrix, calculate_E_matrix,
extract_camerapose, disambiguate_camerapose

from rectification import rectification


start = datetime.datetime.now()



def main():

    for i in range(10):

        img1 = cv2.imread("C:\\Users\\jngo1\\OneDrive - Department of Education and Training\\EE
Work\\Code\\depth-
est\\Dataset\\Cityscapes\\leftImg8bit_trainvaltest\\leftImg8bit\\test\\berlin\\left_image_{}.png".format(st
r(i+1)), 0)

        img2 = cv2.imread("C:\\Users\\jngo1\\OneDrive - Department of Education and Training\\EE
Work\\Code\\depth-
est\\Dataset\\Cityscapes\\rightImg8bit_trainvaltest\\rightImg8bit\\test\\berlin\\right_image_{}.png".form
at(str(i+1)), 0)


        width = int(img1.shape[1]* 0.3) # 0.3

        height = int(img1.shape[0]* 0.3) # 0.3


        img1 = cv2.resize(img1, (width, height), interpolation = cv2.INTER_AREA)

        img2 = cv2.resize(img2, (width, height), interpolation = cv2.INTER_AREA)


        #___________________Camera Parameters_________________________________

        K11 = np.array([[2262.52, 0.00, 1096.98],
```

```
            [0, 2265.30, 513.137],
            [0, 0, 1]])

K12 = np.array([[2262.52, 0.00, 1886.9],
            [0, 2265.30, 513.137],
            [0, 0, 1]])

camera_params = [(K11, K12)]

while(1):
    try:
        list_kp1, list_kp2 = draw_keypoints_and_match(img1, img2)

        #_________________________________Calibration_________________________________

        F = RANSAC_F_matrix([list_kp1, list_kp2])
        # print("F matrix", F)
        # print("=="*20, '\n')
        K1, K2 = camera_params[0]
        E = calculate_E_matrix(F, K1, K2)
        # print("E matrix", E)
        # print("=="*20, '\n')
        camera_poses = extract_camerapose(E)
        best_camera_pose = disambiguate_camerapose(camera_poses, list_kp1)
        # print("Best_Camera_Pose:")
        # print("=="*20)
        # print("Roatation", best_camera_pose[0])
        # print()
        # print("Transaltion", best_camera_pose[1])
        # print("=="*20, '\n')
        pts1 = np.int32(list_kp1)
        pts2 = np.int32(list_kp2)
        #______________________________Rectification__________________________________
```

```
            rectified_pts1, rectified_pts2, img1_rectified, img2_rectified = rectification(img1, img2, pts1, pts2, F)
            break
        except Exception as e:
            # print("error", e)
            continue

    # Find epilines corresponding to points in right image (second image) and drawing its lines on left image

    lines1 = cv2.computeCorrespondEpilines(rectified_pts2.reshape(-1, 1, 2), 2, F)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(img1_rectified, img2_rectified, lines1, rectified_pts1, rectified_pts2)

    # Find epilines corresponding to points in left image (first image) and drawing its lines on right image

    lines2 = cv2.computeCorrespondEpilines(rectified_pts1.reshape(-1, 1, 2), 1, F)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(img2_rectified, img1_rectified, lines2, rectified_pts2, rectified_pts1)

    cv2.imwrite("left_image.png", img5)
    cv2.imwrite("right_image.png", img3)

    #______________________________Correspondance__________________________________

    imgL_rectified = cv2.imread("rectified_1.png", 0)
    imgR_rectified = cv2.imread("rectified_2.png", 0)

    # # # CREATE DEPTH MAP
    win_size = 3
    min_disp = 0
    num_disp = win_size * 16

    stereo = cv2.StereoSGBM_create(
```

```
        minDisparity=min_disp,
        numDisparities=num_disp,
        blockSize=11,
        P1= 8 * 1 * win_size ** 2,
        P2= 16 * 1 * win_size ** 2,
        mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
    )

    disparity = stereo.compute(img1, img2)

    # left_matcher = cv2.StereoBM_create(numDisparities=16, blockSize=11)

    # lmbda=8000
    # sigma=1.5
    # right_matcher = cv2.ximgproc.createRightMatcher(left_matcher)
    # # FILTER Parameters
    # visual_multiplier = 6
    # wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=left_matcher)
    # wls_filter.setLambda(lmbda)

    # wls_filter.setSigmaColor(sigma)
    # displ = left_matcher.compute(imgL_rectified, imgR_rectified)  # .astype(np.float32)/16
    # dispr = right_matcher.compute(imgR_rectified, imgL_rectified)  # .astype(np.float32)/16
    # displ = np.int16(displ)
    # dispr = np.int16(dispr)
    # filteredImg = wls_filter.filter(displ, imgL_rectified, None, dispr)  # important to put "imgL" here!!!

    # filteredImg = cv2.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255,
norm_type=cv2.NORM_MINMAX);
    # filteredImg = np.uint8(filteredImg)

    cv2.imwrite("C:\\Users\\jngo1\\Onedrive - Department of Education and Training\\EE
Work\\Code\\Disparities\\disparity_map_base_{}.png".format(str(i+1)), disparity)
```

```
        baseline = 0.209313
        f = (2262.52 + 2265.302) / 2



if __name__ == "__main__":
    main()

end = datetime.datetime.now()
print(f'Time to compute disparity map using StereoBM', end-start)
```

Calibration Class:

```
import math
import cv2
import random as rd
import numpy as np
import tensorflow as tf  # noqa: E402

from superpoint.settings import EXPER_PATH  # noqa: E402

def draw_keypoints_and_match(img1, img2):
    # """This function is used for finding keypoints and dercriptors in the image and
    #     find best matches using brute force/FLANN based matcher."""

    # Note : Can use sift too to improve feature extraction, but it can be patented again so it could brake the
code in future!
    # Note: ORB is not scale independent so number of keypoints depend on scale
    # Initiate ORB detector
    orb = cv2.ORB_create()
    # find the keypoints and descriptors with ORB
    kp1, des1 = orb.detectAndCompute(img1,None)
    kp2, des2 = orb.detectAndCompute(img2,None)

    #__________________________Brute Force Matcher______________________________
    # create BFMatcher object
```

```
   bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

   # Match descriptors.
   matches = bf.match(des1,des2)

   # Sort them in the order of their distance.
   matches = sorted(matches, key = lambda x:x.distance)
   # print(len(matches))

   # Select first 30 matches.
   final_matches = matches[:30]

   #___________________________SuperPoint___________________________________
  #_________________________SuperPoint Detection and Matching____________________


os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
def extract_superpoint_keypoints_and_descriptors(keypoint_map, descriptor_map,
                                      keep_k_points=1000):

   def select_k_best(points, k):
      """ Select the k most probable points (and strip their proba).
      points has shape (num_points, 3) where the last coordinate is the proba. """
      sorted_prob = points[points[:, 2].argsort(), :2]
      start = min(k, points.shape[0])
      return sorted_prob[-start:, :]

   # Extract keypoints
   keypoints = np.where(keypoint_map > 0)
   prob = keypoint_map[keypoints[0], keypoints[1]]
   keypoints = np.stack([keypoints[0], keypoints[1], prob], axis=-1)

   keypoints = select_k_best(keypoints, keep_k_points)
   keypoints = keypoints.astype(int)
```

```
    # Get descriptors for keypoints
    desc = descriptor_map[keypoints[:, 0], keypoints[:, 1]]

    # Convert from just pts to cv2.KeyPoints
    keypoints = [cv2.KeyPoint(p[1], p[0], 1) for p in keypoints]

    return keypoints, desc
def match_descriptors(kp1, desc1, kp2, desc2):
    # Match the keypoints with the warped_keypoints with nearest neighbor search
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)

    matches = bf.match(desc1, desc2)
    matches_idx = np.array([m.queryIdx for m in matches])

    m_kp1 = [kp1[idx] for idx in matches_idx]
    matches_idx = np.array([m.trainIdx for m in matches])
    m_kp2 = [kp2[idx] for idx in matches_idx]

    return m_kp1, m_kp2, matches



def compute_homography(matched_kp1, matched_kp2):
    matched_pts1 = cv2.KeyPoint_convert(matched_kp1)
    matched_pts2 = cv2.KeyPoint_convert(matched_kp2)
    H, inliers = cv2.findHomography(matched_pts1[:, [1, 0]],
                                    matched_pts2[:, [1, 0]],cv2.RANSAC,5.0)
    inliers = inliers.flatten()
    print(H)
    return H, inliers



def preprocess_image(img_file, img_size):
    img = cv2.imread(img_file, cv2.IMREAD_COLOR)
```

```
    img = cv2.resize(img, img_size)
    img_orig = img.copy()

    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img = np.expand_dims(img, 2)
    img = img.astype(np.float32)
    img_preprocessed = img / 255.

return img_preprocessed, img_orig
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser = argparse.ArgumentParser(description='Compute the homography \
        between two images with the SuperPoint feature matches.')
    parser.add_argument('weights_name', type=str)
    parser.add_argument('img1_path', type=str)
    parser.add_argument('img2_path', type=str)
    parser.add_argument('--H', type=int, default=480,
                help='The height in pixels to resize the images to. \
                    (default: 480)')
    parser.add_argument('--W', type=int, default=640,
                help='The width in pixels to resize the images to. \
                    (default: 640)')
    parser.add_argument('--k_best', type=int, default=1000,
                help='Maximum number of keypoints to keep \
                (default: 1000)')
    args = parser.parse_args()

    weights_name = args.weights_name
    img1_file = args.img1_path
    img2_file = args.img2_path
    img_size = (args.W, args.H)
    keep_k_best = args.k_best

    weights_root_dir = Path(EXPER_PATH, 'saved_models')
```

```
weights_root_dir.mkdir(parents=True, exist_ok=True)
weights_dir = Path(weights_root_dir, weights_name)

graph = tf.Graph()
with tf.Session(graph=graph) as sess:
    tf.saved_model.loader.load(sess,
                      [tf.saved_model.tag_constants.SERVING],
                      str(weights_dir))

    input_img_tensor = graph.get_tensor_by_name('superpoint/image:0')
    output_prob_nms_tensor = graph.get_tensor_by_name('superpoint/prob_nms:0')
    output_desc_tensors = graph.get_tensor_by_name('superpoint/descriptors:0')

    img1, img1_orig = preprocess_image(img1_file, img_size)
    out1 = sess.run([output_prob_nms_tensor, output_desc_tensors],
            feed_dict={input_img_tensor: np.expand_dims(img1, 0)})
    keypoint_map1 = np.squeeze(out1[0])

    descriptor_map1 = np.squeeze(out1[1])
    kp1, desc1 = extract_superpoint_keypoints_and_descriptors(
        keypoint_map1, descriptor_map1, keep_k_best)

    img2, img2_orig = preprocess_image(img2_file, img_size)
    out2 = sess.run([output_prob_nms_tensor, output_desc_tensors],
            feed_dict={input_img_tensor: np.expand_dims(img2, 0)})
    keypoint_map2 = np.squeeze(out2[0])
    descriptor_map2 = np.squeeze(out2[1])
    kp2, desc2 = extract_superpoint_keypoints_and_descriptors(
        keypoint_map2, descriptor_map2, keep_k_best)

    # Match and get rid of outliers

    m_kp1, m_kp2, matches = match_descriptors(kp1, desc1, kp2, desc2)
```

```
    H, inliers= compute_homography(m_kp1, m_kp2)


    matches = np.array(matches)[inliers.astype(bool)].tolist()
    matched_img = cv2.drawMatches(img1_orig, kp1, img2_orig, kp2, matches,
                        None, matchColor=(0, 255, 0),
                        singlePointColor=(0, 0, 255))
#_________________________FLANN based Matcher_________________________________
# FLANN_INDEX_LSH = 6
# index_params = dict(
#     algorithm=FLANN_INDEX_LSH,
#     table_number=6,  # 12
#     key_size=12,  # 20
#     multi_probe_level=1,
# )  # 2
# search_params = dict(checks=0)  # or pass empty dictionary
# flann = cv2.FlannBasedMatcher(index_params, search_params)
# flann_match_pairs = flann.knnMatch(des1, des2, k=2)


# # Filter matches using the Lowe's ratio test
# ratio_threshold = 0.3
# filtered_matches = []
# for m, n in flann_match_pairs:
#     if m.distance < ratio_threshold * n.distance:
#         filtered_matches.append(m)


# print("FMatches", len(filtered_matches))
# final_matches =  filtered_matches[:100]
#_____________________________________________________________________________



# Draw keypoints
```

```
    img_with_keypoints =
cv2.drawMatches(img1,kp1,img2,kp2,final_matches,None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SING
LE_POINTS)
    cv2.imwrite("images_with_matching_keypoints.png", img_with_keypoints)

    # Getting x,y coordinates of the matches
    list_kp1 = [list(kp1[mat.queryIdx].pt) for mat in final_matches]
    list_kp2 = [list(kp2[mat.trainIdx].pt) for mat in final_matches]

    return list_kp1, list_kp2


def calculate_F_matrix(list_kp1, list_kp2):
    """This function is used to calculate the F matrix from a set of 8 points using SVD.
       Furthermore, the rank of F matrix is reduced from 3 to 2 to make the epilines converge."""

    A = np.zeros(shape=(len(list_kp1), 9))

    for i in range(len(list_kp1)):
        x1, y1 = list_kp1[i][0], list_kp1[i][1]
        x2, y2 = list_kp2[i][0], list_kp2[i][1]
        A[i] = np.array([x1*x2, x1*y2, x1, y1*x2, y1*y2, y1, x2, y2, 1])

    U, s, Vt = np.linalg.svd(A)
    F = Vt[-1,:]
    F = F.reshape(3,3)

    # Downgrading the rank of F matrix from 3 to 2
    Uf, Df, Vft = np.linalg.svd(F)
    Df[2] = 0
    s = np.zeros((3,3))
    for i in range(3):
```

```
        s[i][i] = Df[i]

    F = np.dot(Uf, np.dot(s, Vft))
    return F

def RANSAC_F_matrix(list_of_cood_list):
    """This method is used to shortlist the best F matrix using RANSAC based on the number of inliers."""

    list_kp1 = list_of_cood_list[0]
    list_kp2 = list_of_cood_list[1]
    pairs = list(zip(list_kp1, list_kp2))
    max_inliers = 20
    threshold = 0.5  # Tune this value

    for i in range(1000):
        pairs = rd.sample(pairs, 8)
        rd_list_kp1, rd_list_kp2 = zip(*pairs)
        F = calculate_F_matrix(rd_list_kp1, rd_list_kp2)

        tmp_inliers_img1 = []
        tmp_inliers_img2 = []

        for i in range(len(list_kp1)):
            img1_x = np.array([list_kp1[i][0], list_kp1[i][1], 1])
            img2_x = np.array([list_kp2[i][0], list_kp2[i][1], 1])
            distance = abs(np.dot(img2_x.T, np.dot(F,img1_x)))
            # print(distance)

            if distance < threshold:
                tmp_inliers_img1.append(list_kp1[i])
                tmp_inliers_img2.append(list_kp2[i])

        num_of_inliers = len(tmp_inliers_img1)
```

```
        # if num_of_inliers > inlier_count:
        #     inlier_count = num_of_inliers
        #     Best_F = F

        if num_of_inliers > max_inliers:
            print("Number of inliers", num_of_inliers)
            max_inliers = num_of_inliers
            Best_F = F
            inliers_img1 = tmp_inliers_img1
            inliers_img2 = tmp_inliers_img2
            # print("Best F matrix", Best_F)

    return Best_F



def calculate_E_matrix(F, K1, K2):
    """Calculation of Essential matrix"""

    E = np.dot(K2.T, np.dot(F,K1))
    return E



def extract_camerapose(E):
    """This function extracts all the camera pose solutions from the E matrix"""

    U, s, Vt = np.linalg.svd(E)
    W = np.array([[0,-1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])

    C1, C2 = U[:, 2], -U[:, 2]
    R1, R2 = np.dot(U, np.dot(W,Vt)), np.dot(U, np.dot(W.T, Vt))
    # print("C1", C1, "\n", "C2", C2, "\n", "R1", R1, "\n", "R2", R2, "\n")
```

```
    camera_poses = [[R1, C1], [R1, C2], [R2, C1], [R2, C2]]
    return camera_poses



def disambiguate_camerapose(camera_poses, list_kp1):
    """This fucntion is used to find the correct camera pose based on the chirelity condition from all 4
solutions."""

    max_len = 0
    # Calculating 3D points
    for pose in camera_poses:

        front_points = []
        for point in list_kp1:
            # Chirelity check
            X = np.array([point[0], point[1], 1])
            V = X - pose[1]

            condition = np.dot(pose[0][2], V)
            if condition > 0:
                front_points.append(point)

        if len(front_points) > max_len:
            max_len = len(front_points)
            best_camera_pose =  pose

    return best_camera_pose



def drawlines(img1src, img2src, lines, pts1src, pts2src):
    """This fucntion is used to visualize the epilines on the images
        img1 - image on which we draw the epilines for the points in img2
        lines - corresponding epilines """
    r, c = img1src.shape
```

```
    img1color = cv2.cvtColor(img1src, cv2.COLOR_GRAY2BGR)
    img2color = cv2.cvtColor(img2src, cv2.COLOR_GRAY2BGR)
    # Edit: use the same random seed so that two images are comparable!
    np.random.seed(0)
    for r, pt1, pt2 in zip(lines, pts1src, pts2src):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, -r[2]/r[1]])
        x1, y1 = map(int, [c, -(r[2]+r[0]*c)/r[1]])
        img1color = cv2.line(img1color, (x0, y0), (x1, y1), color, 1)
        img1color = cv2.circle(img1color, tuple(pt1), 5, color, -1)
        img2color = cv2.circle(img2color, tuple(pt2), 5, color, -1)

    return img1color, img2color
```

Dataset Handler:

```
import os

folder_path = "C:\\Users\\jngo1\\OneDrive - Department of Education and Training\\EE Work\\Code\\depth-est\\Dataset\\Cityscapes\\leftImg8bit_trainvaltest\\leftImg8bit\\test\\berlin"

for i in range(100):
  old_name = "berlin_0{}_000019_leftImg8bit.png".format(str(i).zfill(5))
  old_path = os.path.join(folder_path, old_name)
  new_name = "left_image_{}.png".format(str(i+1))
  new_path = os.path.join(folder_path, new_name)
  os.rename(old_path, new_path)

  print('Done!')
```

Rectification Class:

```
import numpy as np
import cv2

def rectification(img1, img2, pts1, pts2, F):
    # """This function is used to rectify the images to make camera pose's parallel and thus make epiplines as horizontal.
```

```
    #     Since camera distortion parameters are not given we will use cv2.stereoRectifyUncalibrated(),
instead of stereoRectify().
    # """


    # Stereo rectification
    h1, w1 = img1.shape
    h2, w2 = img2.shape


    _, H1, H2 = cv2.stereoRectifyUncalibrated(np.float32(pts1), np.float32(pts2), F, imgSize=(w1, h1))
    print("H1",H1)
    print("H2",H2)


    rectified_pts1 = np.zeros((pts1.shape), dtype=int)
    rectified_pts2 = np.zeros((pts2.shape), dtype=int)


    # Rectify the feature points
    for i in range(pts1.shape[0]):
        source1 = np.array([pts1[i][0], pts1[i][1], 1])
        new_point1 = np.dot(H1, source1)
        new_point1[0] = int(new_point1[0]/new_point1[2])
        new_point1[1] = int(new_point1[1]/new_point1[2])
        new_point1 = np.delete(new_point1, 2)
        rectified_pts1[i] = new_point1


        source2 = np.array([pts2[i][0], pts2[i][1], 1])
        new_point2 = np.dot(H2, source2)
        new_point2[0] = int(new_point2[0]/new_point2[2])
        new_point2[1] = int(new_point2[1]/new_point2[2])
        new_point2 = np.delete(new_point2, 2)
        rectified_pts2[i] = new_point2


    # Rectify the images and save them
    img1_rectified = cv2.warpPerspective(img1, H1, (w1, h1))
    img2_rectified = cv2.warpPerspective(img2, H2, (w2, h2))
```

```
   cv2.imwrite("rectified_1.png", img1_rectified)
   cv2.imwrite("rectified_2.png", img2_rectified)

   return rectified_pts1, rectified_pts2, img1_rectified, img2_rectified
```

Appendix B: Code for ORB Class Reference (Savani, 2022)

```
orb = cv2.ORB_create()
   # find the keypoints and descriptors with ORB
   kp1, des1 = orb.detectAndCompute(img1,None)
   kp2, des2 = orb.detectAndCompute(img2,None)
```

Appendix C: Code for SuperPoint Algorithm (Magic Leap, 2020) (https://github.com/magicleap/SuperPointPretrainedNetwork)

```
import argparse
import glob
import numpy as np
import os
import time

import cv2
import torch

# Stub to warn about opencv version.
if int(cv2.__version__[0]) < 3: # pragma: no cover
  print('Warning: OpenCV 3 is not installed')

# Jet colormap for visualization.
myjet = np.array([[0.        , 0.        , 0.5       ],
                  [0.        , 0.        , 0.99910873],
                  [0.        , 0.37843137, 1.        ],
                  [0.        , 0.83333333, 1.        ],
                  [0.30044276, 1.        , 0.66729918],
                  [0.66729918, 1.        , 0.30044276],
                  [1.        , 0.90123457, 0.        ],
```

```
        [1.        , 0.48002905, 0.        ],
        [0.99910873, 0.07334786, 0.        ],
        [0.5       , 0.        , 0.        ]])

class SuperPointNet(torch.nn.Module):
  """ Pytorch definition of SuperPoint Network. """
  def __init__(self):
    super(SuperPointNet, self).__init__()
    self.relu = torch.nn.ReLU(inplace=True)
    self.pool = torch.nn.MaxPool2d(kernel_size=2, stride=2)
    c1, c2, c3, c4, c5, d1 = 64, 64, 128, 128, 256, 256
    # Shared Encoder.
    self.conv1a = torch.nn.Conv2d(1, c1, kernel_size=3, stride=1, padding=1)
    self.conv1b = torch.nn.Conv2d(c1, c1, kernel_size=3, stride=1, padding=1)
    self.conv2a = torch.nn.Conv2d(c1, c2, kernel_size=3, stride=1, padding=1)
    self.conv2b = torch.nn.Conv2d(c2, c2, kernel_size=3, stride=1, padding=1)
    self.conv3a = torch.nn.Conv2d(c2, c3, kernel_size=3, stride=1, padding=1)
    self.conv3b = torch.nn.Conv2d(c3, c3, kernel_size=3, stride=1, padding=1)
    self.conv4a = torch.nn.Conv2d(c3, c4, kernel_size=3, stride=1, padding=1)
    self.conv4b = torch.nn.Conv2d(c4, c4, kernel_size=3, stride=1, padding=1)
    # Detector Head.
    self.convPa = torch.nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
    self.convPb = torch.nn.Conv2d(c5, 65, kernel_size=1, stride=1, padding=0)
    # Descriptor Head.
    self.convDa = torch.nn.Conv2d(c4, c5, kernel_size=3, stride=1, padding=1)
    self.convDb = torch.nn.Conv2d(c5, d1, kernel_size=1, stride=1, padding=0)

  def forward(self, x):
    """ Forward pass that jointly computes unprocessed point and descriptor
    tensors.
    Input
      x: Image pytorch tensor shaped N x 1 x H x W.
    Output
      semi: Output point pytorch tensor shaped N x 65 x H/8 x W/8.
```

```
  desc: Output descriptor pytorch tensor shaped N x 256 x H/8 x W/8.
 """
 # Shared Encoder.
 x = self.relu(self.conv1a(x))
 x = self.relu(self.conv1b(x))
 x = self.pool(x)
 x = self.relu(self.conv2a(x))
 x = self.relu(self.conv2b(x))
 x = self.pool(x)
 x = self.relu(self.conv3a(x))
 x = self.relu(self.conv3b(x))
 x = self.pool(x)
 x = self.relu(self.conv4a(x))
 x = self.relu(self.conv4b(x))
 # Detector Head.
 cPa = self.relu(self.convPa(x))
 semi = self.convPb(cPa)
 # Descriptor Head.
 cDa = self.relu(self.convDa(x))
 desc = self.convDb(cDa)
 dn = torch.norm(desc, p=2, dim=1) # Compute the norm.
 desc = desc.div(torch.unsqueeze(dn, 1)) # Divide by norm to normalize.
 return semi, desc


class SuperPointFrontend(object):
 """ Wrapper around pytorch net to help with pre and post image processing. """
 def __init__(self, weights_path, nms_dist, conf_thresh, nn_thresh,
        cuda=False):
  self.name = 'SuperPoint'
  self.cuda = cuda
  self.nms_dist = nms_dist
  self.conf_thresh = conf_thresh
  self.nn_thresh = nn_thresh # L2 descriptor distance for good match.
```

```
  self.cell = 8 # Size of each output cell. Keep this fixed.
  self.border_remove = 4 # Remove points this close to the border.

  # Load the network in inference mode.
  self.net = SuperPointNet()
  if cuda:
    # Train on GPU, deploy on GPU.
    self.net.load_state_dict(torch.load(weights_path))
    self.net = self.net.cuda()
  else:
    # Train on GPU, deploy on CPU.
    self.net.load_state_dict(torch.load(weights_path,
                             map_location=lambda storage, loc: storage))
  self.net.eval()

def nms_fast(self, in_corners, H, W, dist_thresh):
  """
  Run a faster approximate Non-Max-Suppression on numpy corners shaped:
    3xN [x_i,y_i,conf_i]^T

  Algo summary: Create a grid sized HxW. Assign each corner location a 1, rest
  are zeros. Iterate through all the 1's and convert them either to -1 or 0.
  Suppress points by setting nearby values to 0.

  Grid Value Legend:
  -1 : Kept.
   0 : Empty or suppressed.
   1 : To be processed (converted to either kept or supressed).

  NOTE: The NMS first rounds points to integers, so NMS distance might not
  be exactly dist_thresh. It also assumes points are within image boundaries.

  Inputs
    in_corners - 3xN numpy array with corners [x_i, y_i, confidence_i]^T.
```

```
  H - Image height.
  W - Image width.
  dist_thresh - Distance to suppress, measured as an infinty norm distance.
Returns
  nmsed_corners - 3xN numpy matrix with surviving corners.
  nmsed_inds - N length numpy vector with surviving corner indices.
"""
grid = np.zeros((H, W)).astype(int) # Track NMS data.
inds = np.zeros((H, W)).astype(int) # Store indices of points.
# Sort by confidence and round to nearest int.
inds1 = np.argsort(-in_corners[2,:])
corners = in_corners[:,inds1]
rcorners = corners[:2,:].round().astype(int) # Rounded corners.
# Check for edge case of 0 or 1 corners.
if rcorners.shape[1] == 0:
  return np.zeros((3,0)).astype(int), np.zeros(0).astype(int)
if rcorners.shape[1] == 1:
  out = np.vstack((rcorners, in_corners[2])).reshape(3,1)
  return out, np.zeros((1)).astype(int)
# Initialize the grid.
for i, rc in enumerate(rcorners.T):
  grid[rcorners[1,i], rcorners[0,i]] = 1
  inds[rcorners[1,i], rcorners[0,i]] = i
# Pad the border of the grid, so that we can NMS points near the border.
pad = dist_thresh
grid = np.pad(grid, ((pad,pad), (pad,pad)), mode='constant')
# Iterate through points, highest to lowest conf, suppress neighborhood.
count = 0
for i, rc in enumerate(rcorners.T):
  # Account for top and left padding.
  pt = (rc[0]+pad, rc[1]+pad)
  if grid[pt[1], pt[0]] == 1: # If not yet suppressed.
    grid[pt[1]-pad:pt[1]+pad+1, pt[0]-pad:pt[0]+pad+1] = 0
    grid[pt[1], pt[0]] = -1
```

```
      count += 1
    # Get all surviving -1's and return sorted array of remaining corners.
    keepy, keepx = np.where(grid==-1)
    keepy, keepx = keepy - pad, keepx - pad
    inds_keep = inds[keepy, keepx]
    out = corners[:, inds_keep]
    values = out[-1, :]
    inds2 = np.argsort(-values)
    out = out[:, inds2]
    out_inds = inds1[inds_keep[inds2]]
    return out, out_inds

  def run(self, img):
    """ Process a numpy image to extract points and descriptors.
    Input
      img - HxW numpy float32 input image in range [0,1].
    Output
      corners - 3xN numpy array with corners [x_i, y_i, confidence_i]^T.
      desc - 256xN numpy array of corresponding unit normalized descriptors.
      heatmap - HxW numpy heatmap in range [0,1] of point confidences.
      """
    assert img.ndim == 2, 'Image must be grayscale.'
    assert img.dtype == np.float32, 'Image must be float32.'
    H, W = img.shape[0], img.shape[1]
    inp = img.copy()
    inp = (inp.reshape(1, H, W))
    inp = torch.from_numpy(inp)
    inp = torch.autograd.Variable(inp).view(1, 1, H, W)
    if self.cuda:
      inp = inp.cuda()
    # Forward pass of network.
    outs = self.net.forward(inp)
    semi, coarse_desc = outs[0], outs[1]
    # Convert pytorch -> numpy.
```

```
semi = semi.data.cpu().numpy().squeeze()
# --- Process points.
dense = np.exp(semi) # Softmax.
dense = dense / (np.sum(dense, axis=0)+.00001) # Should sum to 1.
# Remove dustbin.
nodust = dense[:-1, :, :]
# Reshape to get full resolution heatmap.
Hc = int(H / self.cell)
Wc = int(W / self.cell)
nodust = nodust.transpose(1, 2, 0)
heatmap = np.reshape(nodust, [Hc, Wc, self.cell, self.cell])
heatmap = np.transpose(heatmap, [0, 2, 1, 3])
heatmap = np.reshape(heatmap, [Hc*self.cell, Wc*self.cell])
xs, ys = np.where(heatmap >= self.conf_thresh) # Confidence threshold.
if len(xs) == 0:
  return np.zeros((3, 0)), None, None
pts = np.zeros((3, len(xs))) # Populate point data sized 3xN.
pts[0, :] = ys
pts[1, :] = xs
pts[2, :] = heatmap[xs, ys]
pts, _ = self.nms_fast(pts, H, W, dist_thresh=self.nms_dist) # Apply NMS.
inds = np.argsort(pts[2,:])
pts = pts[:,inds[::-1]] # Sort by confidence.
# Remove points along border.
bord = self.border_remove
toremoveW = np.logical_or(pts[0, :] < bord, pts[0, :] >= (W-bord))
toremoveH = np.logical_or(pts[1, :] < bord, pts[1, :] >= (H-bord))
toremove = np.logical_or(toremoveW, toremoveH)
pts = pts[:, ~toremove]
# --- Process descriptor.
D = coarse_desc.shape[1]
if pts.shape[1] == 0:
  desc = np.zeros((D, 0))
else:
```

```
    # Interpolate into descriptor map using 2D point locations.
    samp_pts = torch.from_numpy(pts[:2, :].copy())
    samp_pts[0, :] = (samp_pts[0, :] / (float(W)/2.)) - 1.
    samp_pts[1, :] = (samp_pts[1, :] / (float(H)/2.)) - 1.
    samp_pts = samp_pts.transpose(0, 1).contiguous()
    samp_pts = samp_pts.view(1, 1, -1, 2)
    samp_pts = samp_pts.float()
    if self.cuda:
      samp_pts = samp_pts.cuda()
    desc = torch.nn.functional.grid_sample(coarse_desc, samp_pts)
    desc = desc.data.cpu().numpy().reshape(D, -1)
    desc /= np.linalg.norm(desc, axis=0)[np.newaxis, :]
  return pts, desc, heatmap


class PointTracker(object):
 """ Class to manage a fixed memory of points and descriptors that enables
 sparse optical flow point tracking.

 Internally, the tracker stores a 'tracks' matrix sized M x (2+L), of M
 tracks with maximum length L, where each row corresponds to:
 row_m = [track_id_m, avg_desc_score_m, point_id_0_m, ..., point_id_L-1_m].
 """

 def __init__(self, max_length, nn_thresh):
  if max_length < 2:
   raise ValueError('max_length must be greater than or equal to 2.')
  self.maxl = max_length
  self.nn_thresh = nn_thresh
  self.all_pts = []
  for n in range(self.maxl):
   self.all_pts.append(np.zeros((2, 0)))
  self.last_desc = None
  self.tracks = np.zeros((0, self.maxl+2))
```

```
    self.track_count = 0
    self.max_score = 9999

  def nn_match_two_way(self, desc1, desc2, nn_thresh):
    """
    Performs two-way nearest neighbor matching of two sets of descriptors, such
    that the NN match from descriptor A->B must equal the NN match from B->A.

    Inputs:
      desc1 - NxM numpy matrix of N corresponding M-dimensional descriptors.
      desc2 - NxM numpy matrix of N corresponding M-dimensional descriptors.
      nn_thresh - Optional descriptor distance below which is a good match.

    Returns:
      matches - 3xL numpy array, of L matches, where L <= N and each column i is
                a match of two descriptors, d_i in image 1 and d_j' in image 2:
                [d_i index, d_j' index, match_score]^T
    """
    assert desc1.shape[0] == desc2.shape[0]
    if desc1.shape[1] == 0 or desc2.shape[1] == 0:
      return np.zeros((3, 0))
    if nn_thresh < 0.0:
      raise ValueError('\'nn_thresh\' should be non-negative')
    # Compute L2 distance. Easy since vectors are unit normalized.
    dmat = np.dot(desc1.T, desc2)
    dmat = np.sqrt(2-2*np.clip(dmat, -1, 1))
    # Get NN indices and scores.
    idx = np.argmin(dmat, axis=1)
    scores = dmat[np.arange(dmat.shape[0]), idx]
    # Threshold the NN matches.
    keep = scores < nn_thresh
    # Check if nearest neighbor goes both directions and keep those.
    idx2 = np.argmin(dmat, axis=0)
    keep_bi = np.arange(len(idx)) == idx2[idx]
```

```
  keep = np.logical_and(keep, keep_bi)
  idx = idx[keep]
  scores = scores[keep]
  # Get the surviving point indices.
  m_idx1 = np.arange(desc1.shape[1])[keep]
  m_idx2 = idx
  # Populate the final 3xN match data structure.
  matches = np.zeros((3, int(keep.sum())))
  matches[0, :] = m_idx1
  matches[1, :] = m_idx2
  matches[2, :] = scores
  return matches

def get_offsets(self):
  """ Iterate through list of points and accumulate an offset value. Used to
  index the global point IDs into the list of points.

  Returns
    offsets - N length array with integer offset locations.
  """
  # Compute id offsets.
  offsets = []
  offsets.append(0)
  for i in range(len(self.all_pts)-1): # Skip last camera size, not needed.
    offsets.append(self.all_pts[i].shape[1])
  offsets = np.array(offsets)
  offsets = np.cumsum(offsets)
  return offsets

def update(self, pts, desc):
  """ Add a new set of point and descriptor observations to the tracker.

  Inputs
    pts - 3xN numpy array of 2D point observations.
```

```
  desc - DxN numpy array of corresponding D dimensional descriptors.
"""
if pts is None or desc is None:
  print('PointTracker: Warning, no points were added to tracker.')
  return
assert pts.shape[1] == desc.shape[1]
# Initialize last_desc.
if self.last_desc is None:
  self.last_desc = np.zeros((desc.shape[0], 0))
# Remove oldest points, store its size to update ids later.
remove_size = self.all_pts[0].shape[1]
self.all_pts.pop(0)
self.all_pts.append(pts)
# Remove oldest point in track.
self.tracks = np.delete(self.tracks, 2, axis=1)
# Update track offsets.
for i in range(2, self.tracks.shape[1]):
  self.tracks[:, i] -= remove_size
self.tracks[:, 2:][self.tracks[:, 2:] < -1] = -1
offsets = self.get_offsets()
# Add a new -1 column.
self.tracks = np.hstack((self.tracks, -1*np.ones((self.tracks.shape[0], 1))))
# Try to append to existing tracks.
matched = np.zeros((pts.shape[1])).astype(bool)
matches = self.nn_match_two_way(self.last_desc, desc, self.nn_thresh)
for match in matches.T:
  # Add a new point to it's matched track.
  id1 = int(match[0]) + offsets[-2]
  id2 = int(match[1]) + offsets[-1]
  found = np.argwhere(self.tracks[:, -2] == id1)
  if found.shape[0] > 0:
    matched[int(match[1])] = True
    row = int(found)
    self.tracks[row, -1] = id2
```

```
    if self.tracks[row, 1] == self.max_score:
      # Initialize track score.
      self.tracks[row, 1] = match[2]
    else:
      # Update track score with running average.
      # NOTE(dd): this running average can contain scores from old matches
      #           not contained in last max_length track points.
      track_len = (self.tracks[row, 2:] != -1).sum() - 1.
      frac = 1. / float(track_len)
      self.tracks[row, 1] = (1.-frac)*self.tracks[row, 1] + frac*match[2]
  # Add unmatched tracks.
  new_ids = np.arange(pts.shape[1]) + offsets[-1]
  new_ids = new_ids[~matched]
  new_tracks = -1*np.ones((new_ids.shape[0], self.maxl + 2))
  new_tracks[:, -1] = new_ids
  new_num = new_ids.shape[0]
  new_trackids = self.track_count + np.arange(new_num)
  new_tracks[:, 0] = new_trackids
  new_tracks[:, 1] = self.max_score*np.ones(new_ids.shape[0])
  self.tracks = np.vstack((self.tracks, new_tracks))
  self.track_count += new_num # Update the track count.
  # Remove empty tracks.
  keep_rows = np.any(self.tracks[:, 2:] >= 0, axis=1)
  self.tracks = self.tracks[keep_rows, :]
  # Store the last descriptors.
  self.last_desc = desc.copy()
  return

 def get_tracks(self, min_length):
  """ Retrieve point tracks of a given minimum length.
  Input
    min_length - integer >= 1 with minimum track length
  Output
    returned_tracks - M x (2+L) sized matrix storing track indices, where
```

```
    M is the number of tracks and L is the maximum track length.
  """
  if min_length < 1:
    raise ValueError('\'min_length\' too small.')
  valid = np.ones((self.tracks.shape[0])).astype(bool)
  good_len = np.sum(self.tracks[:, 2:] != -1, axis=1) >= min_length
  # Remove tracks which do not have an observation in most recent frame.
  not_headless = (self.tracks[:, -1] != -1)
  keepers = np.logical_and.reduce((valid, good_len, not_headless))
  returned_tracks = self.tracks[keepers, :].copy()
  return returned_tracks


def draw_tracks(self, out, tracks):
  """ Visualize tracks all overlayed on a single image.
  Inputs
    out - numpy uint8 image sized HxWx3 upon which tracks are overlayed.
    tracks - M x (2+L) sized matrix storing track info.
  """
  # Store the number of points per camera.
  pts_mem = self.all_pts
  N = len(pts_mem) # Number of cameras/images.
  # Get offset ids needed to reference into pts_mem.
  offsets = self.get_offsets()
  # Width of track and point circles to be drawn.
  stroke = 1
  # Iterate through each track and draw it.
  for track in tracks:
    clr = myjet[int(np.clip(np.floor(track[1]*10), 0, 9)), :]*255
    for i in range(N-1):
      if track[i+2] == -1 or track[i+3] == -1:
        continue
      offset1 = offsets[i]
      offset2 = offsets[i+1]
      idx1 = int(track[i+2]-offset1)
```

```
      idx2 = int(track[i+3]-offset2)
      pt1 = pts_mem[i][:2, idx1]
      pt2 = pts_mem[i+1][:2, idx2]
      p1 = (int(round(pt1[0])), int(round(pt1[1])))
      p2 = (int(round(pt2[0])), int(round(pt2[1])))
      cv2.line(out, p1, p2, clr, thickness=stroke, lineType=16)
      # Draw end points of each track.
      if i == N-2:
        clr2 = (255, 0, 0)
        cv2.circle(out, p2, stroke, clr2, -1, lineType=16)


class VideoStreamer(object):
  """ Class to help process image streams. Three types of possible inputs:"
    1.) USB Webcam.
    2.) A directory of images (files in directory matching 'img_glob').
    3.) A video file, such as an .mp4 or .avi file.
  """
  def __init__(self, basedir, camid, height, width, skip, img_glob):
    self.cap = []
    self.camera = False
    self.video_file = False
    self.listing = []
    self.sizer = [height, width]
    self.i = 0
    self.skip = skip
    self.maxlen = 1000000
    # If the "basedir" string is the word camera, then use a webcam.
    if basedir == "camera/" or basedir == "camera":
      print('==> Processing Webcam Input.')
      self.cap = cv2.VideoCapture(camid)
      self.listing = range(0, self.maxlen)
      self.camera = True
    else:
      # Try to open as a video.
```

```
    self.cap = cv2.VideoCapture(basedir)
    lastbit = basedir[-4:len(basedir)]
    if (type(self.cap) == list or not self.cap.isOpened()) and (lastbit == '.mp4'):
      raise IOError('Cannot open movie file')
    elif type(self.cap) != list and self.cap.isOpened() and (lastbit != '.txt'):
      print('==> Processing Video Input.')
      num_frames = int(self.cap.get(cv2.CAP_PROP_FRAME_COUNT))
      self.listing = range(0, num_frames)
      self.listing = self.listing[::self.skip]
      self.camera = True
      self.video_file = True
      self.maxlen = len(self.listing)
    else:
      print('==> Processing Image Directory Input.')
      search = os.path.join(basedir, img_glob)
      self.listing = glob.glob(search)
      self.listing.sort()
      self.listing = self.listing[::self.skip]
      self.maxlen = len(self.listing)
      if self.maxlen == 0:
        raise IOError('No images were found (maybe bad \'--img_glob\' parameter?)')

  def read_image(self, impath, img_size):
    """ Read image as grayscale and resize to img_size.
    Inputs
      impath: Path to input image.
      img_size: (W, H) tuple specifying resize size.
    Returns
      grayim: float32 numpy array sized H x W with values in range [0, 1].
    """
    grayim = cv2.imread(impath, 0)
    if grayim is None:
      raise Exception('Error reading image %s' % impath)
    # Image is resized via opencv.
```

```
    interp = cv2.INTER_AREA
    grayim = cv2.resize(grayim, (img_size[1], img_size[0]), interpolation=interp)
    grayim = (grayim.astype('float32') / 255.)
    return grayim

  def next_frame(self):
    """ Return the next frame, and increment internal counter.
    Returns
       image: Next H x W image.
       status: True or False depending whether image was loaded.
    """
    if self.i == self.maxlen:
      return (None, False)
    if self.camera:
      ret, input_image = self.cap.read()
      if ret is False:
        print('VideoStreamer: Cannot get image from camera (maybe bad --camid?)')
        return (None, False)
      if self.video_file:
        self.cap.set(cv2.CAP_PROP_POS_FRAMES, self.listing[self.i])
      input_image = cv2.resize(input_image, (self.sizer[1], self.sizer[0]),
                               interpolation=cv2.INTER_AREA)
      input_image = cv2.cvtColor(input_image, cv2.COLOR_RGB2GRAY)
      input_image = input_image.astype('float')/255.0
    else:
      image_file = self.listing[self.i]
      input_image = self.read_image(image_file, self.sizer)
    # Increment internal counter.
    self.i = self.i + 1
    input_image = input_image.astype('float32')
    return (input_image, True)

if __name__ == '__main__':
```

```
# Parse command line arguments.
parser = argparse.ArgumentParser(description='PyTorch SuperPoint Demo.')
parser.add_argument('input', type=str, default='',
   help='Image directory or movie file or "camera" (for webcam).')
parser.add_argument('--weights_path', type=str, default='superpoint_v1.pth',
   help='Path to pretrained weights file (default: superpoint_v1.pth).')
parser.add_argument('--img_glob', type=str, default='*.png',
   help='Glob match if directory of images is specified (default: \'*.png\').')
parser.add_argument('--skip', type=int, default=1,
   help='Images to skip if input is movie or directory (default: 1).')
parser.add_argument('--show_extra', action='store_true',
   help='Show extra debug outputs (default: False).')
parser.add_argument('--H', type=int, default=120,
   help='Input image height (default: 120).')
parser.add_argument('--W', type=int, default=160,
   help='Input image width (default:160).')
parser.add_argument('--display_scale', type=int, default=2,
   help='Factor to scale output visualization (default: 2).')
parser.add_argument('--min_length', type=int, default=2,
   help='Minimum length of point tracks (default: 2).')
parser.add_argument('--max_length', type=int, default=5,
   help='Maximum length of point tracks (default: 5).')
parser.add_argument('--nms_dist', type=int, default=4,
   help='Non Maximum Suppression (NMS) distance (default: 4).')
parser.add_argument('--conf_thresh', type=float, default=0.015,
   help='Detector confidence threshold (default: 0.015).')
parser.add_argument('--nn_thresh', type=float, default=0.7,
   help='Descriptor matching threshold (default: 0.7).')
parser.add_argument('--camid', type=int, default=0,
   help='OpenCV webcam video capture ID, usually 0 or 1 (default: 0).')
parser.add_argument('--waitkey', type=int, default=1,
   help='OpenCV waitkey time in ms (default: 1).')
parser.add_argument('--cuda', action='store_true',
   help='Use cuda GPU to speed up network processing speed (default: False)')
```

```
parser.add_argument('--no_display', action='store_true',
   help='Do not display images to screen. Useful if running remotely (default: False).')
parser.add_argument('--write', action='store_true',
   help='Save output frames to a directory (default: False)')
parser.add_argument('--write_dir', type=str, default='tracker_outputs/',
   help='Directory where to write output frames (default: tracker_outputs/).')
opt = parser.parse_args()
print(opt)

# This class helps load input images from different sources.
vs = VideoStreamer(opt.input, opt.camid, opt.H, opt.W, opt.skip, opt.img_glob)

print('==> Loading pre-trained network.')
# This class runs the SuperPoint network and processes its outputs.
fe = SuperPointFrontend(weights_path=opt.weights_path,
                        nms_dist=opt.nms_dist,
                        conf_thresh=opt.conf_thresh,
                        nn_thresh=opt.nn_thresh,
                        cuda=opt.cuda)
print('==> Successfully loaded pre-trained network.')

# This class helps merge consecutive point matches into tracks.
tracker = PointTracker(opt.max_length, nn_thresh=fe.nn_thresh)

# Create a window to display the demo.
if not opt.no_display:
  win = 'SuperPoint Tracker'
  cv2.namedWindow(win)
else:
  print('Skipping visualization, will not show a GUI.')

# Font parameters for visualizaton.
font = cv2.FONT_HERSHEY_DUPLEX
font_clr = (255, 255, 255)
```

```
font_pt = (4, 12)
font_sc = 0.4

# Create output directory if desired.
if opt.write:
  print('==> Will write outputs to %s' % opt.write_dir)
  if not os.path.exists(opt.write_dir):
    os.makedirs(opt.write_dir)

print('==> Running Demo.')
while True:

  start = time.time()

  # Get a new image.
  img, status = vs.next_frame()
  if status is False:
    break

  # Get points and descriptors.
  start1 = time.time()
  pts, desc, heatmap = fe.run(img)
  end1 = time.time()

  # Add points and descriptors to the tracker.
  tracker.update(pts, desc)

  # Get tracks for points which were match successfully across all frames.
  tracks = tracker.get_tracks(opt.min_length)

  # Primary output - Show point tracks overlayed on top of input image.
  out1 = (np.dstack((img, img, img)) * 255.).astype('uint8')
  tracks[:, 1] /= float(fe.nn_thresh) # Normalize track scores to [0,1].
  tracker.draw_tracks(out1, tracks)
```

```
if opt.show_extra:
  cv2.putText(out1, 'Point Tracks', font_pt, font, font_sc, font_clr, lineType=16)

# Extra output -- Show current point detections.
out2 = (np.dstack((img, img, img)) * 255.).astype('uint8')
for pt in pts.T:
  pt1 = (int(round(pt[0])), int(round(pt[1])))
  cv2.circle(out2, pt1, 1, (0, 255, 0), -1, lineType=16)
cv2.putText(out2, 'Raw Point Detections', font_pt, font, font_sc, font_clr, lineType=16)

# Extra output -- Show the point confidence heatmap.
if heatmap is not None:
  min_conf = 0.001
  heatmap[heatmap < min_conf] = min_conf
  heatmap = -np.log(heatmap)
  heatmap = (heatmap - heatmap.min()) / (heatmap.max() - heatmap.min() + .00001)
  out3 = myjet[np.round(np.clip(heatmap*10, 0, 9)).astype('int'), :]
  out3 = (out3*255).astype('uint8')
else:
  out3 = np.zeros_like(out2)
cv2.putText(out3, 'Raw Point Confidences', font_pt, font, font_sc, font_clr, lineType=16)

# Resize final output.
if opt.show_extra:
  out = np.hstack((out1, out2, out3))
  out = cv2.resize(out, (3*opt.display_scale*opt.W, opt.display_scale*opt.H))
else:
  out = cv2.resize(out1, (opt.display_scale*opt.W, opt.display_scale*opt.H))

# Display visualization image to screen.
if not opt.no_display:
  cv2.imshow(win, out)
  key = cv2.waitKey(opt.waitkey) & 0xFF
  if key == ord('q'):
```

```
      print('Quitting, \'q\' pressed.')
      break

    # Optionally write images to disk.
    if opt.write:
      out_file = os.path.join(opt.write_dir, 'frame_%05d.png' % vs.i)
      print('Writing image to %s' % out_file)
      cv2.imwrite(out_file, out)

    end = time.time()
    net_t = (1./ float(end1 - start))
    total_t = (1./ float(end - start))
    if opt.show_extra:
      print('Processed image %d (net+post_process: %.2f FPS, total: %.2f FPS).'\
          % (vs.i, net_t, total_t))

  # Close any remaining windows.
  cv2.destroyAllWindows()

  print('==> Finshed Demo.')
```

Appendix D: Evaluation Code

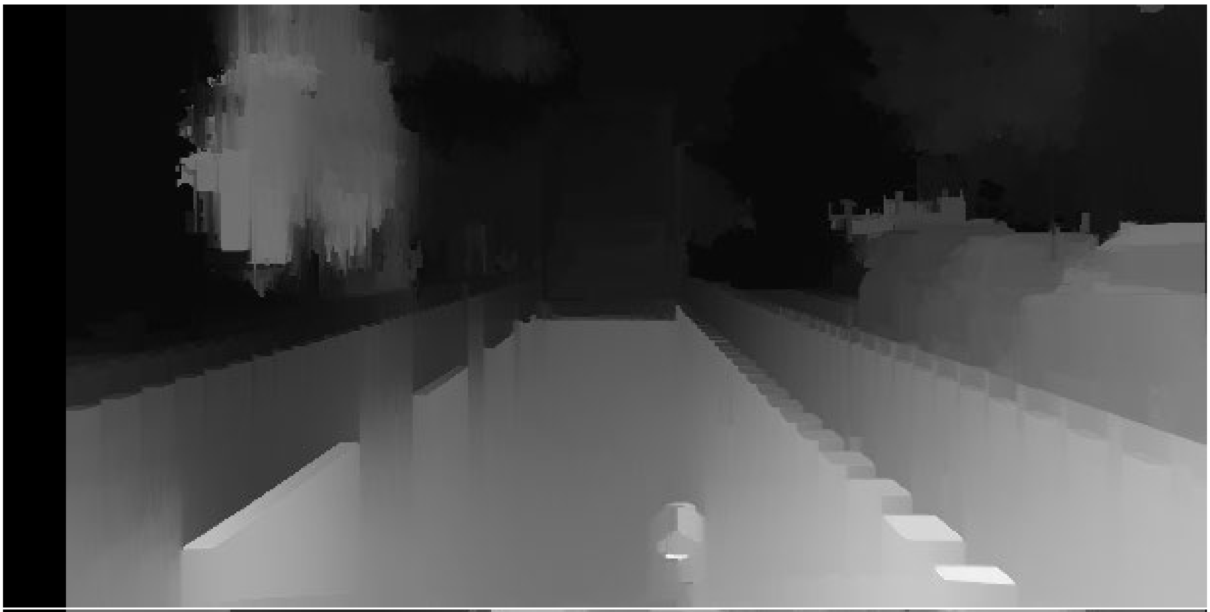
```
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt
import numpy as np
import cv2

def rmse(imageA, imageB):
        # the 'Mean Squared Error' between the two images is the
        # sum of the squared difference between the two images;
        # NOTE: the two images must have the same dimension
    err = np.sum(np.sqrt((imageB.astype("float") - imageA.astype("float"))**2))
    err /= float(imageA.shape[0] * imageA.shape[1])
    return err
```

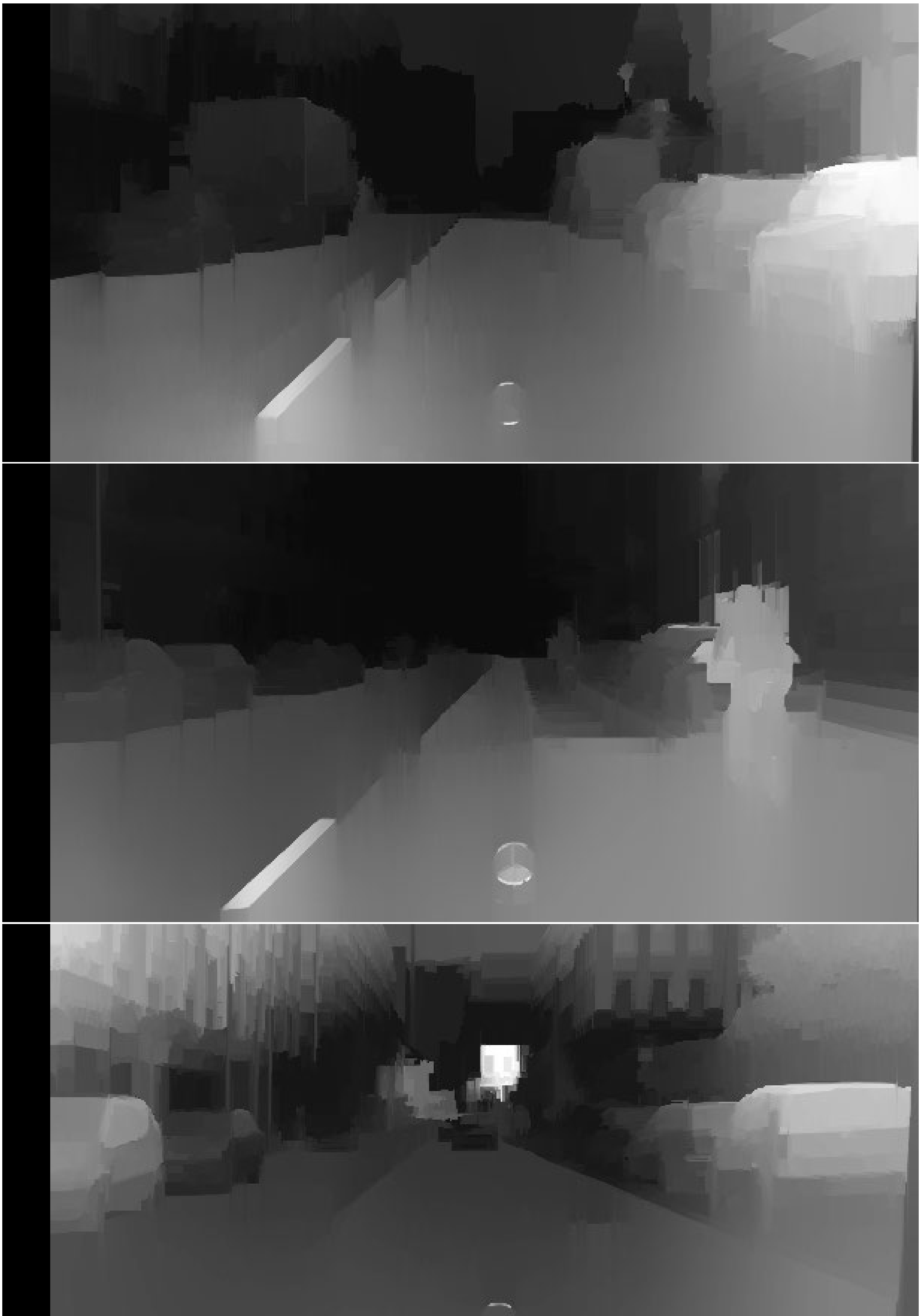
```
def absrel(imageA, imageB):
    err2 = np.sum(imageB.astype("float") - imageA.astype("float"))
    err2 /= float(imageA.shape[0] * imageA.shape[1])
    return err2
        # return the MSE, the lower the error, the more "similar"
        # the two images are

def compare_images(imageA, imageB, title):
    # compute the mean squared error and structural similarity
    # index for the images
    m = rmse(imageA, imageB)
    s = ssim(imageA, imageB)
    r = mae(imageA, imageB)
    # setup the figure
    fig = plt.figure(title)
    plt.suptitle("RMSE: %.2f, SSIM: %.2f, MAE: %.2f" % (m, s, r))
    # show first image
    ax = fig.add_subplot(1, 2, 1)
    plt.imshow(imageA, cmap = plt.cm.gray)
    plt.axis("off")
    # show the second image
    ax = fig.add_subplot(1, 2, 2)
    plt.imshow(imageB, cmap = plt.cm.gray)
    plt.axis("off")
    # show the images
    plt.show()
# load the images -- the original, the original + contrast,
# and the original + computed
gt = cv2.imread("/content/berlin_000099_000019_disparity.png", 0)
computed = cv2.imread("/content/disparity_map_ORB_WLS_100.png", 0)

height, width = computed.shape[:2]

gt1 = cv2.resize(gt, (width, height), interpolation = cv2.INTER_CUBIC)
```

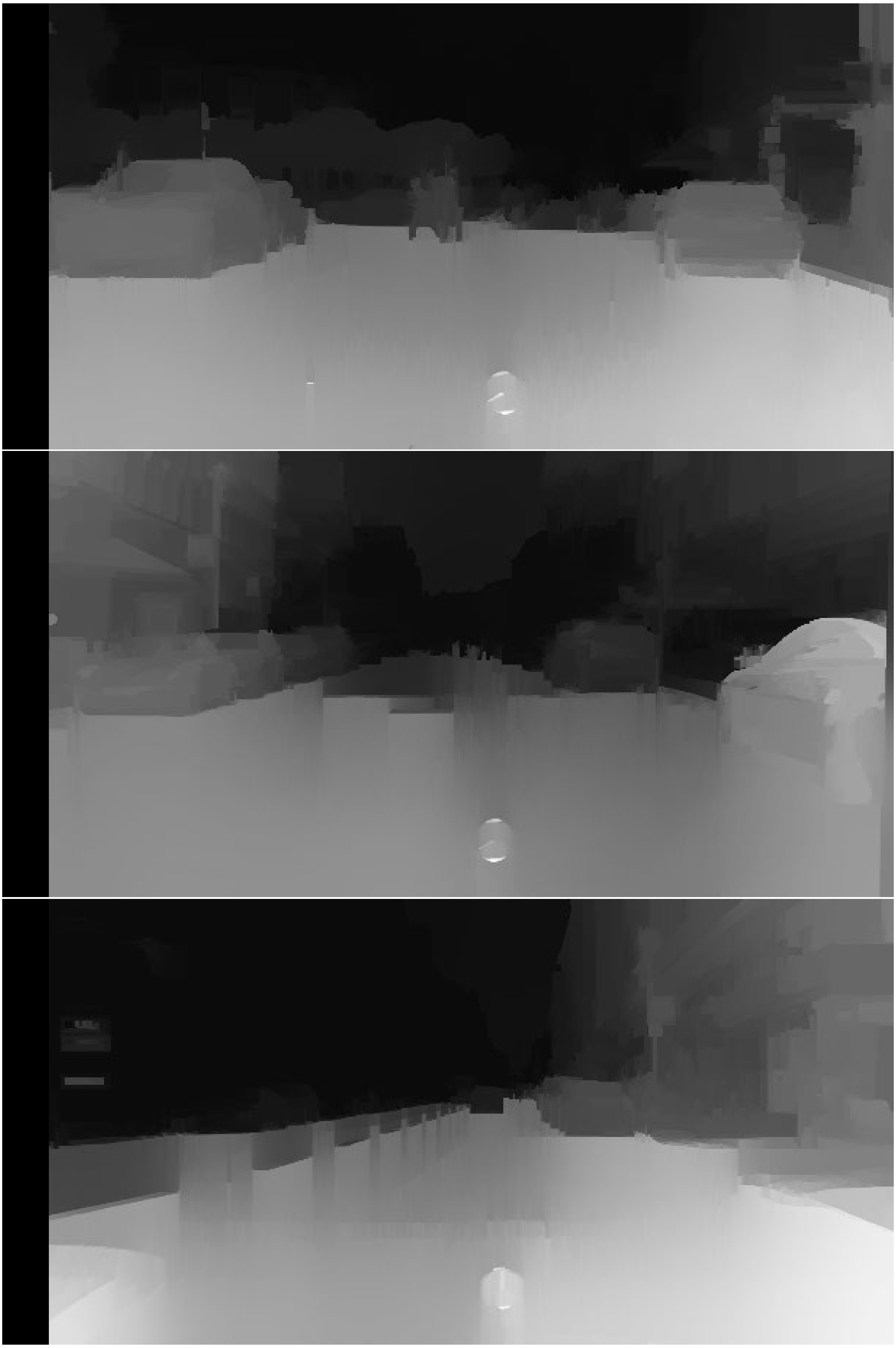
```
cv2_imshow(gt1)
cv2_imshow(computed)


print(img_np.shape)
print(computed.shape)
```

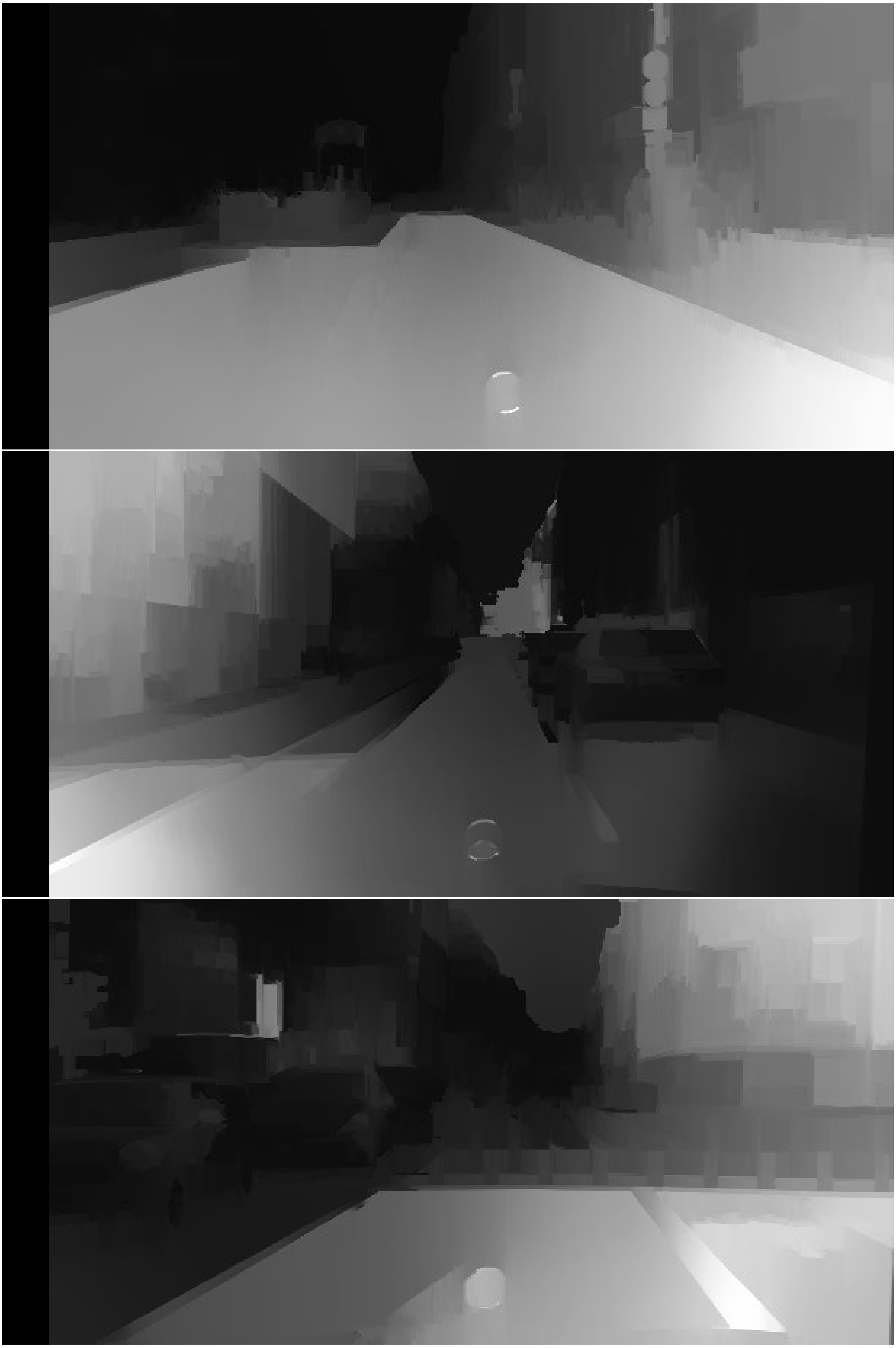
Appendix E: Raw Data

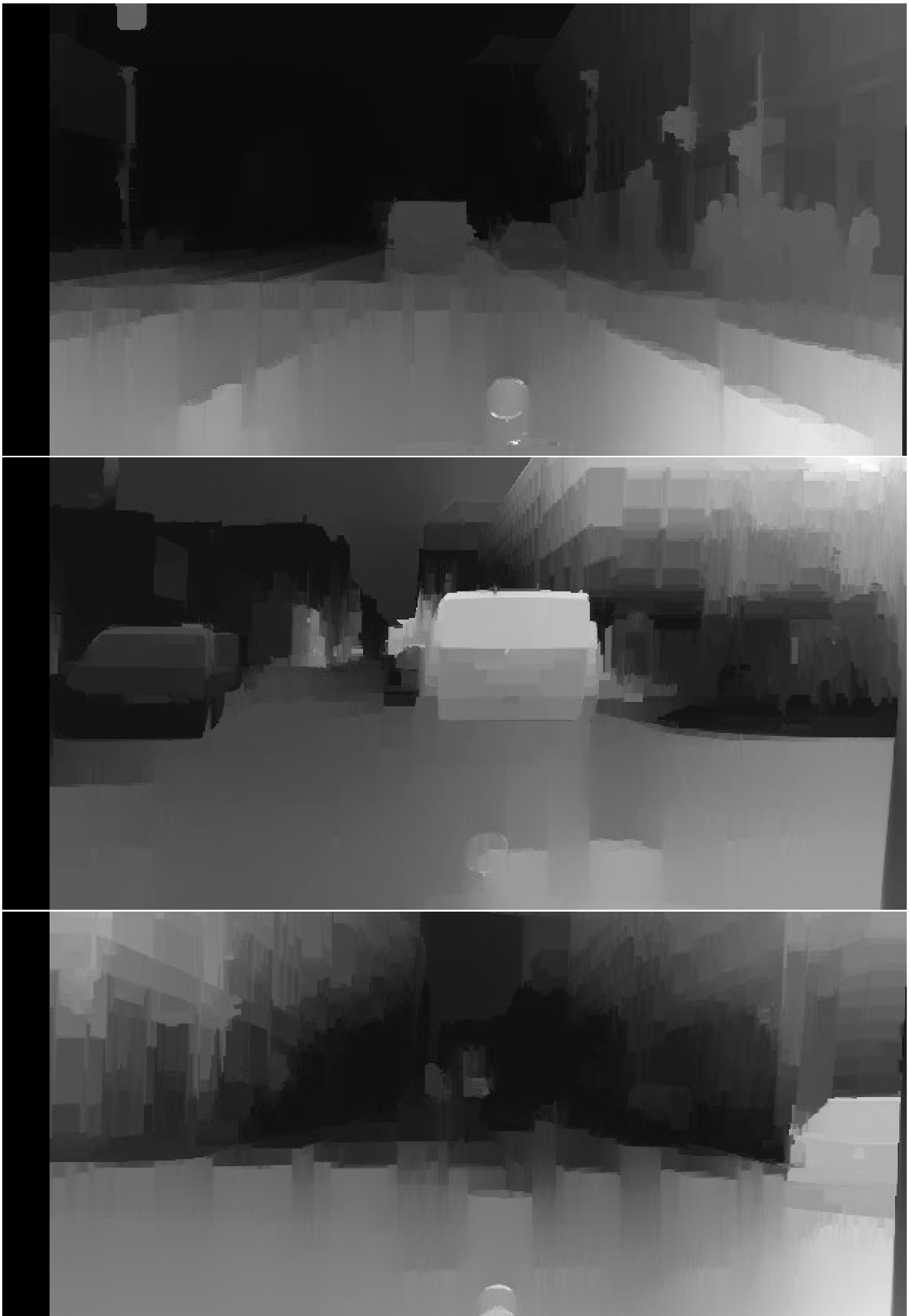
Disparity Maps Generated from the Third Trial of Berlin Dataset:

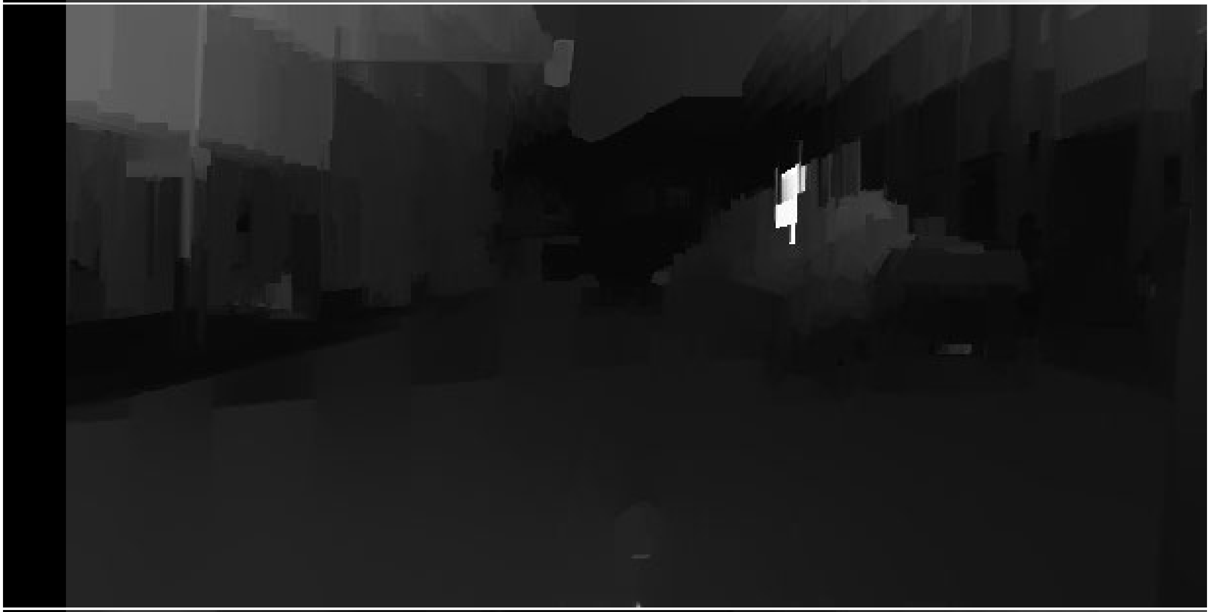


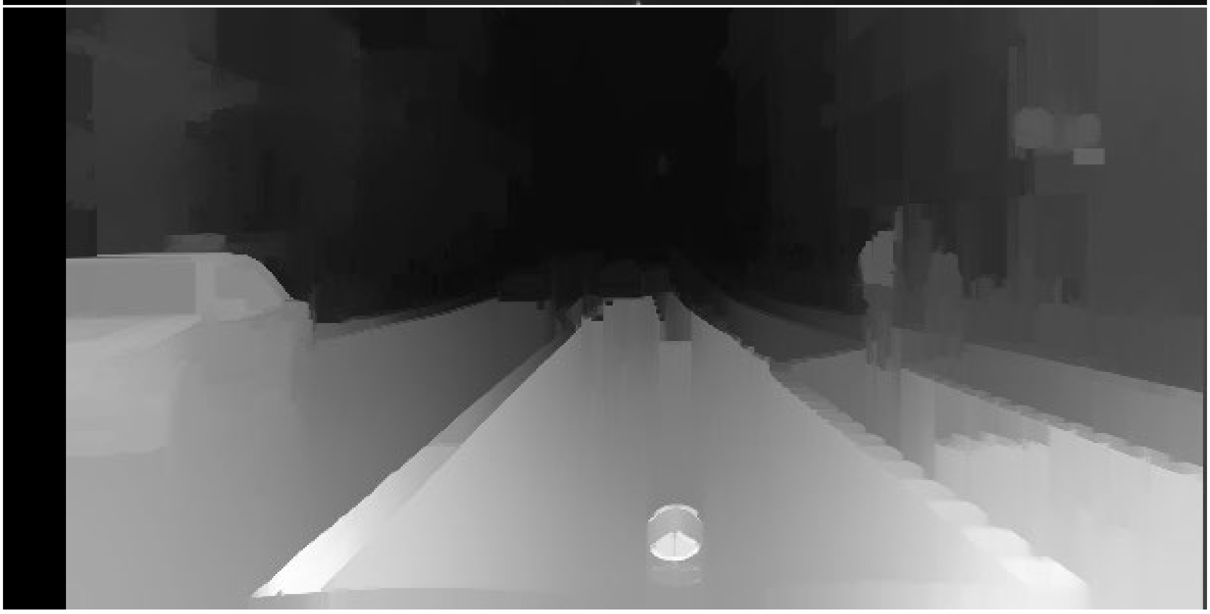


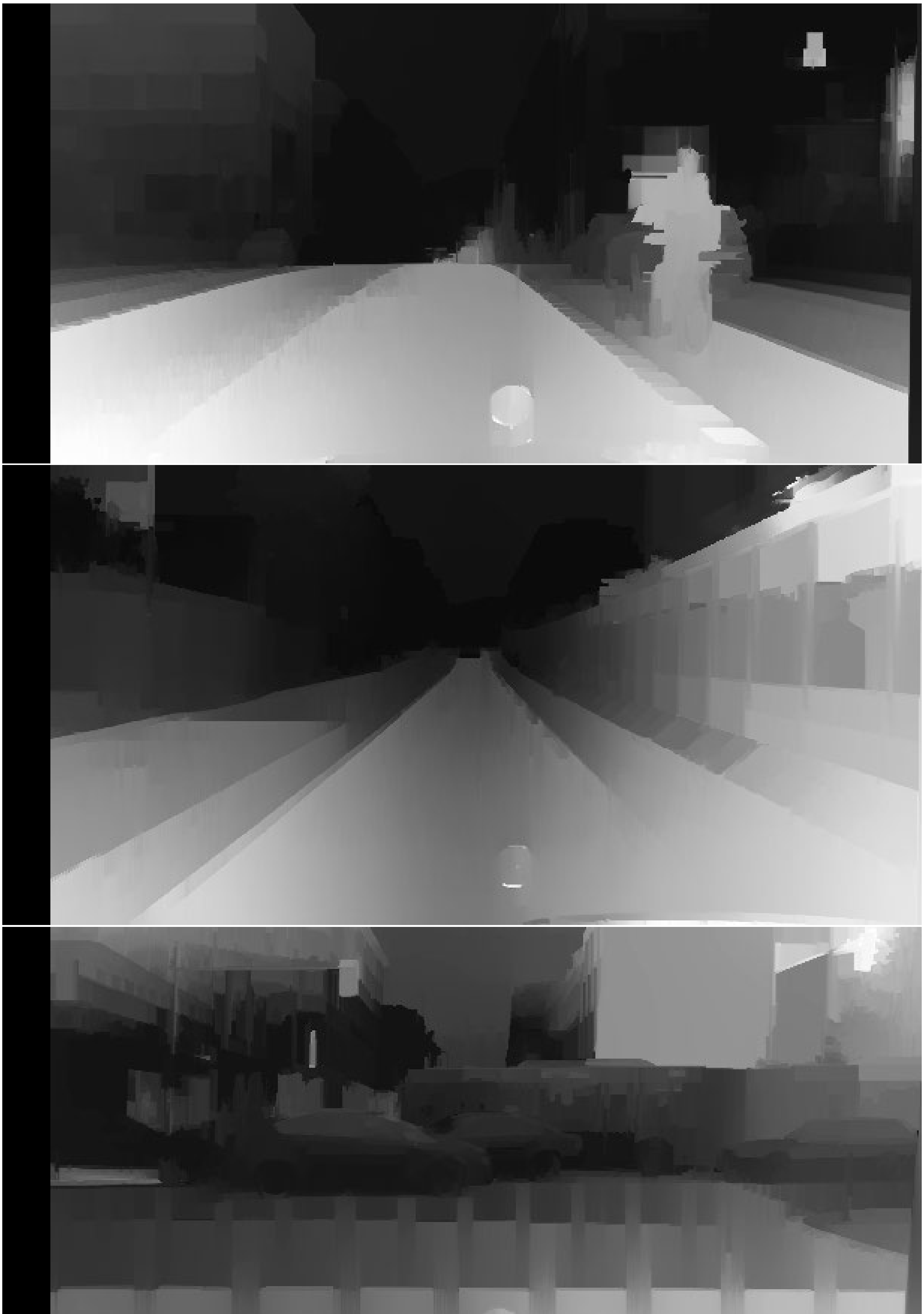


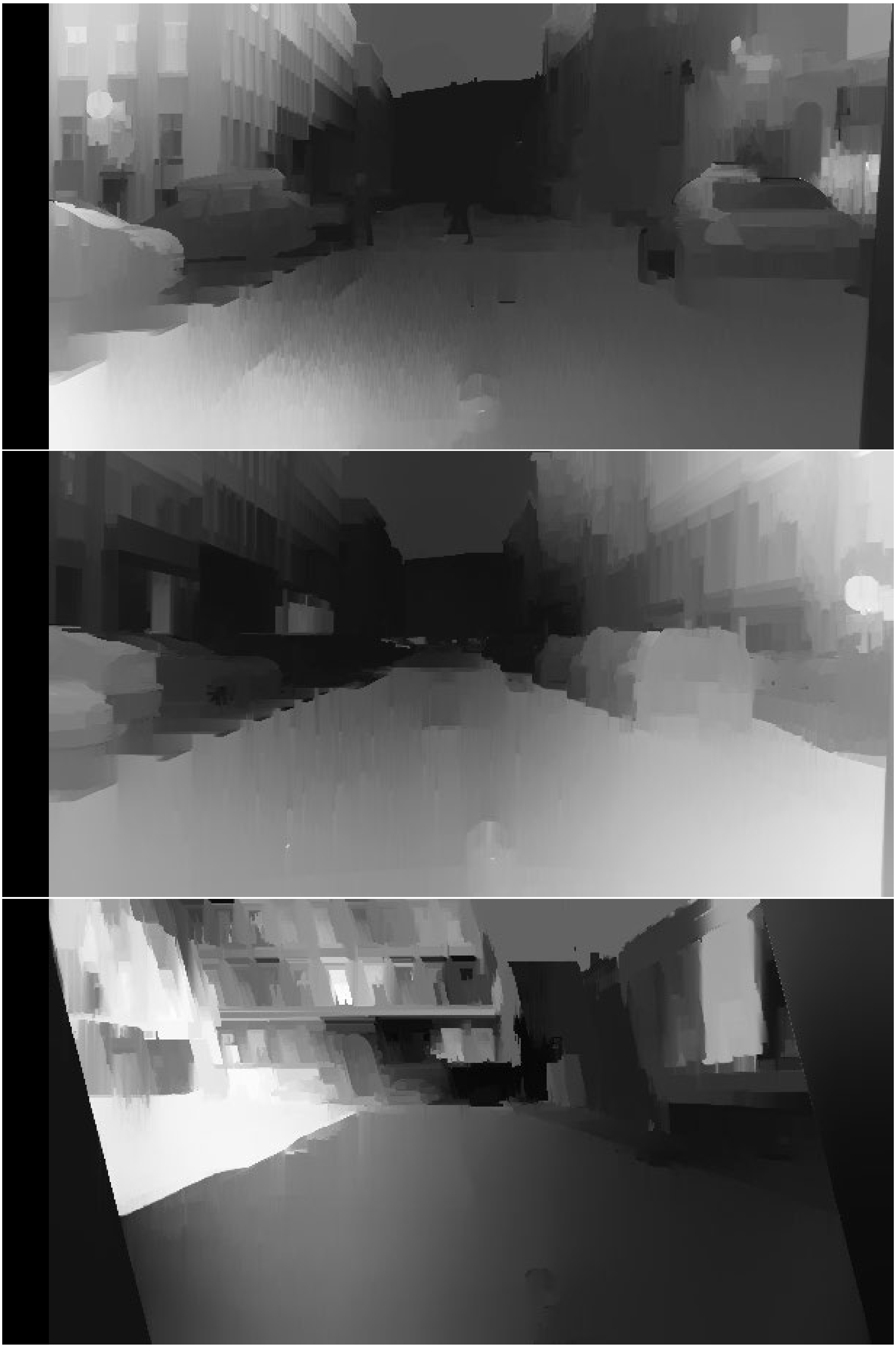


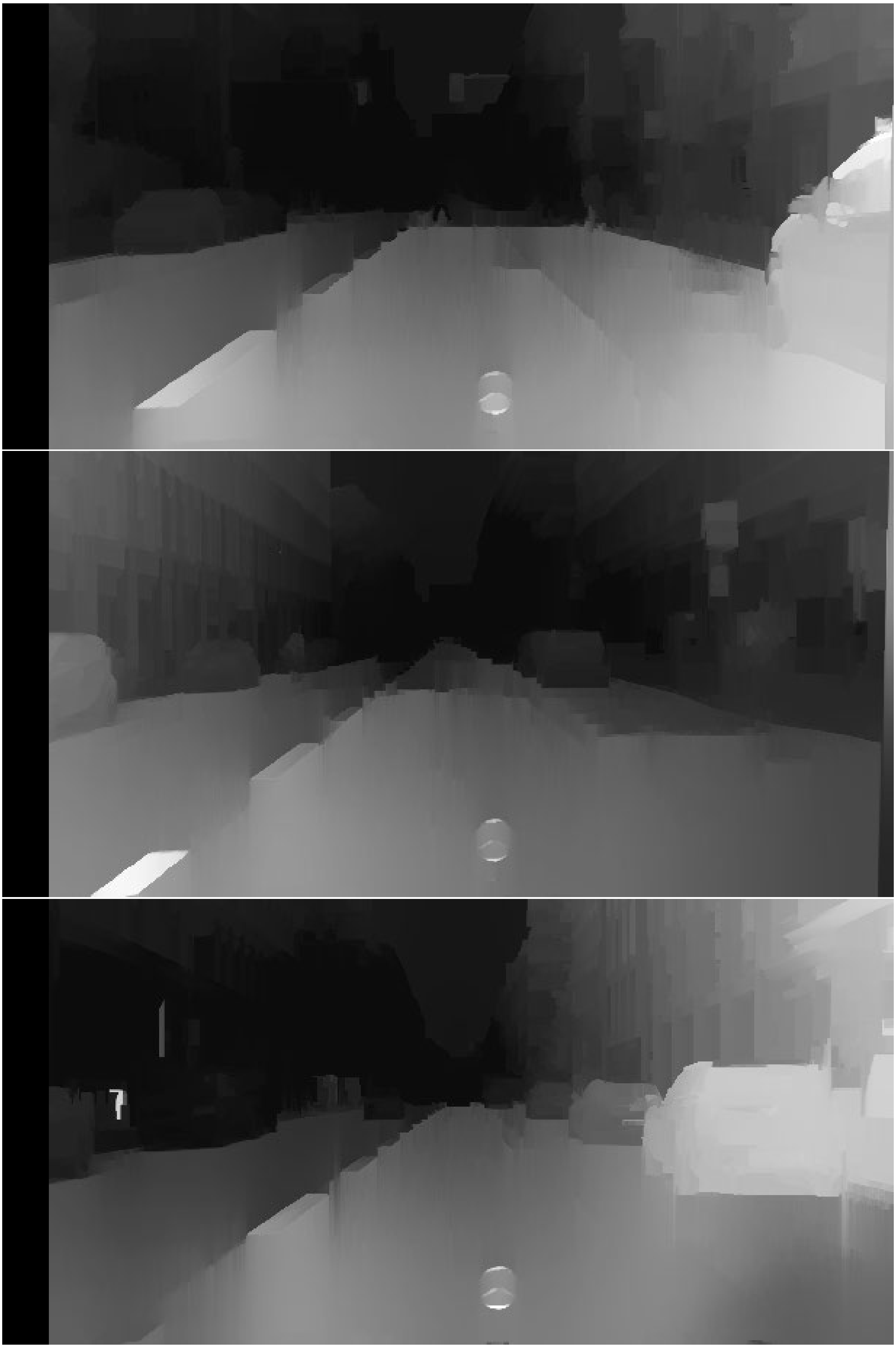


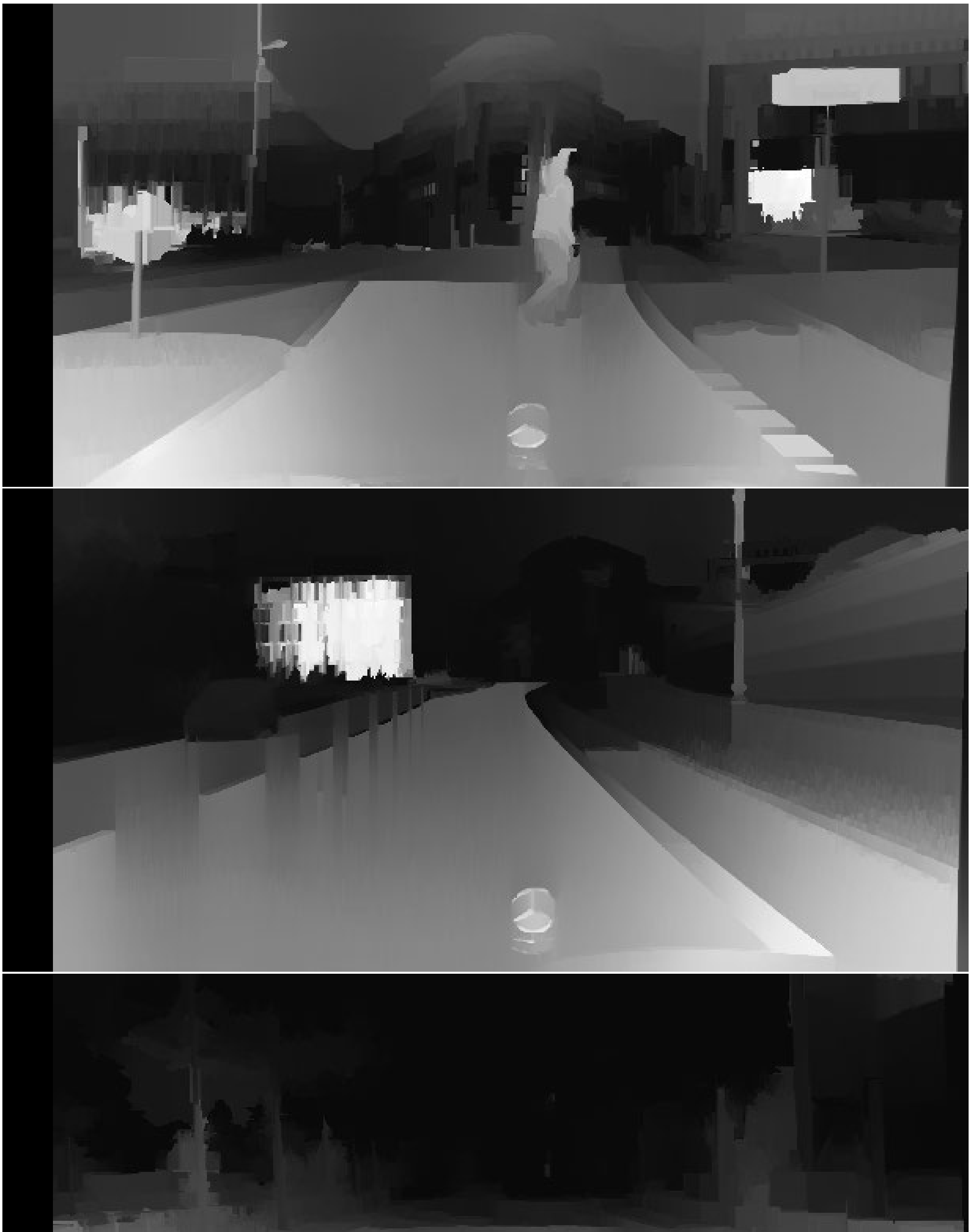


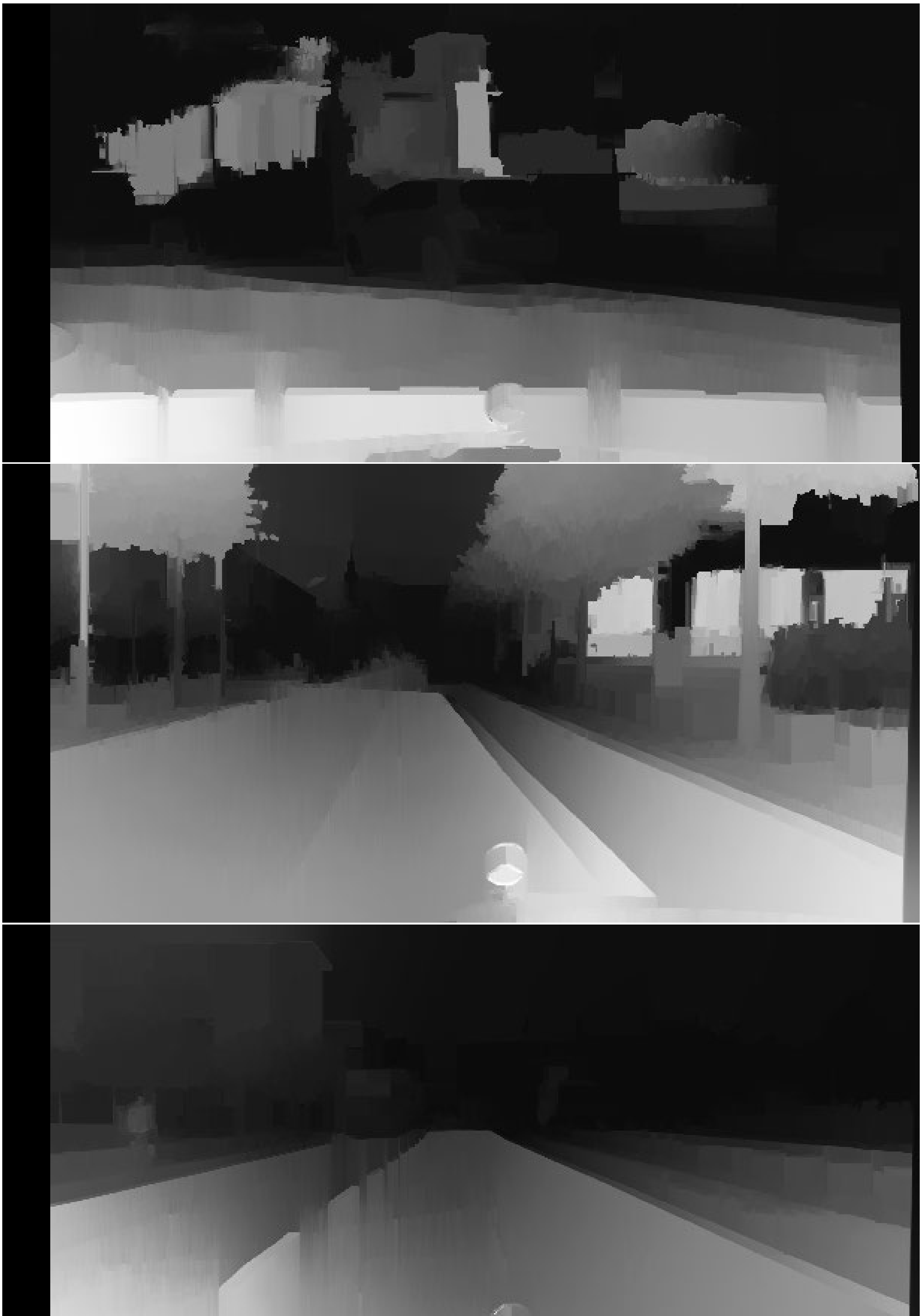


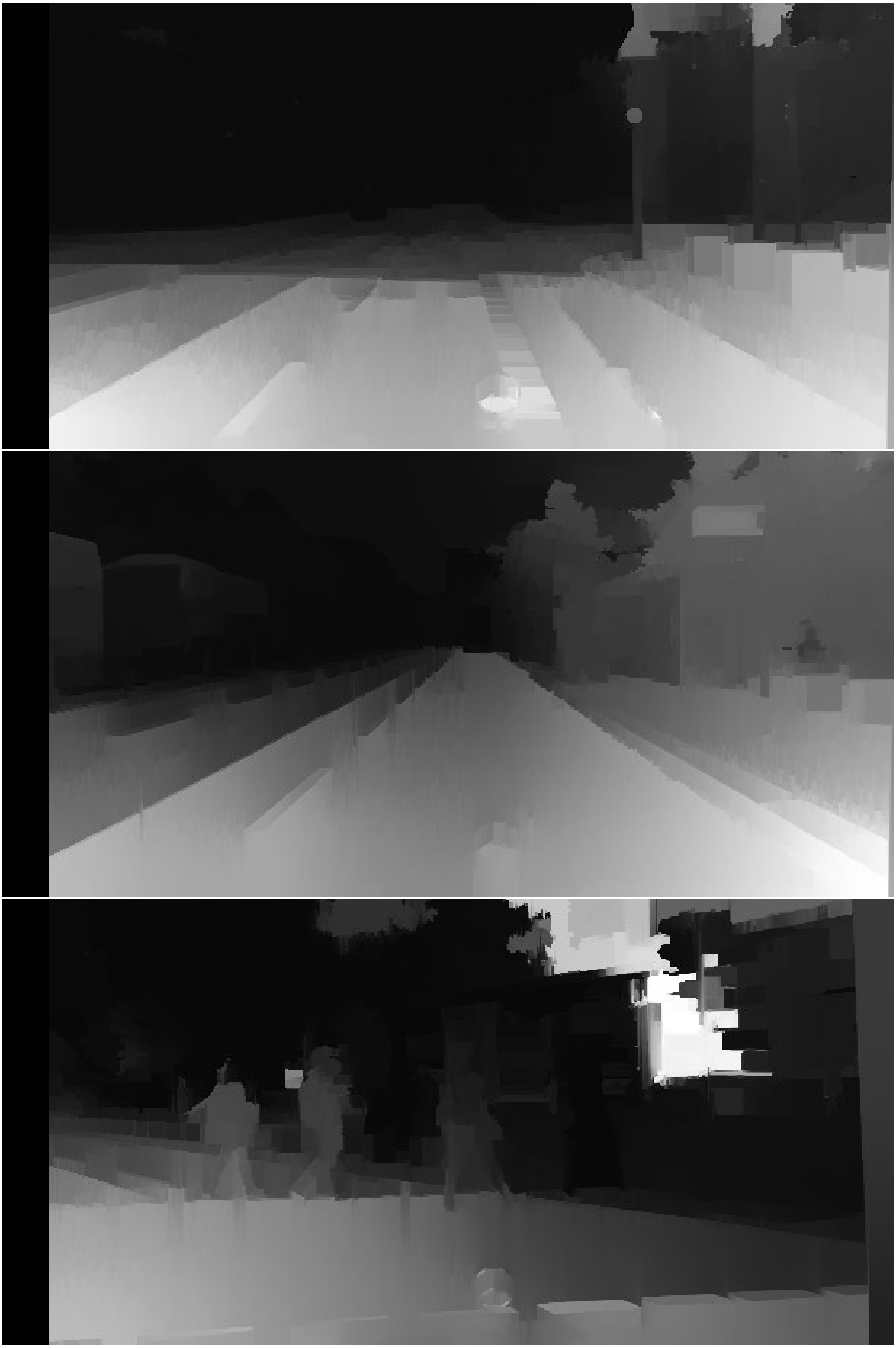


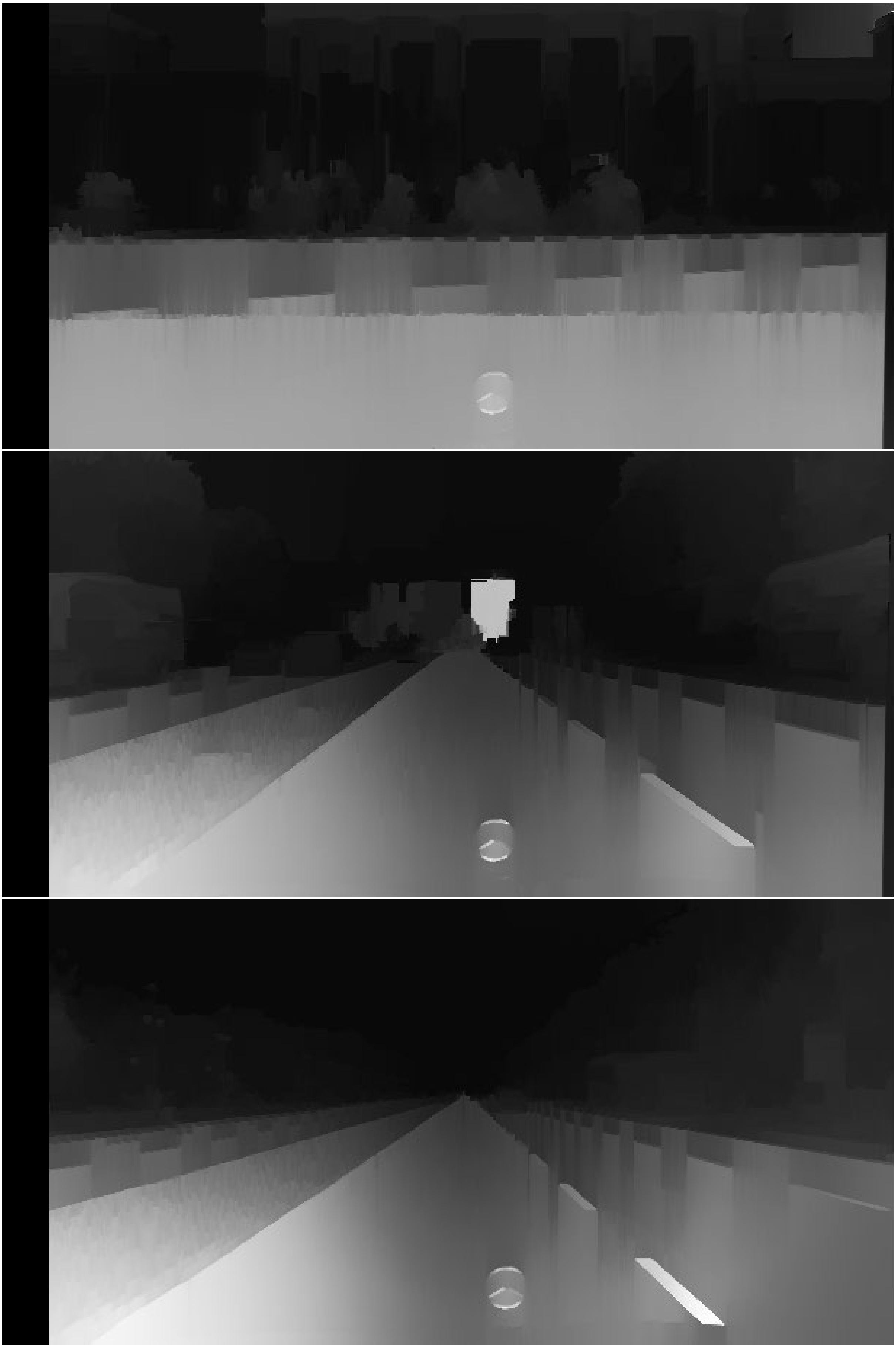


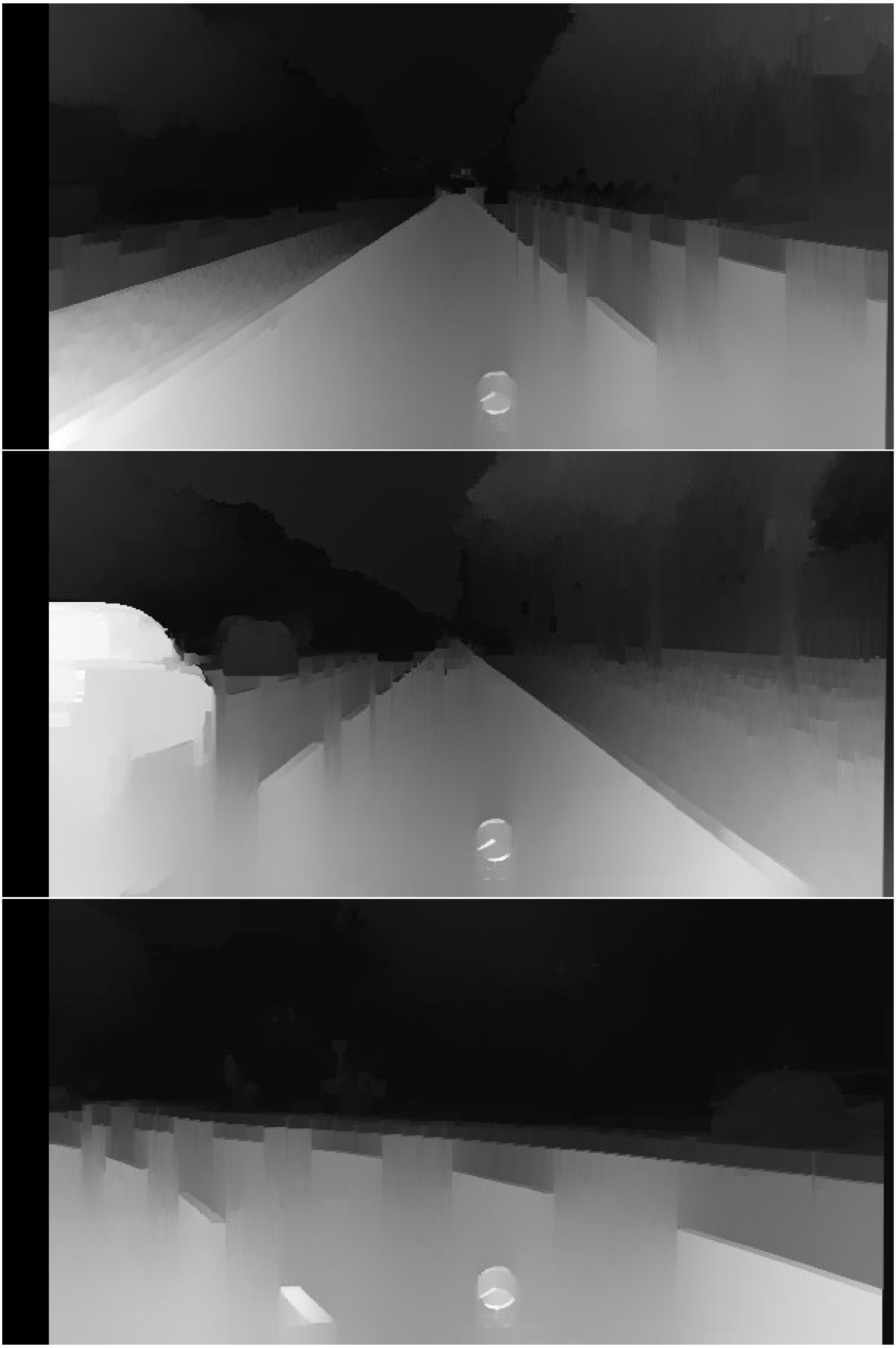


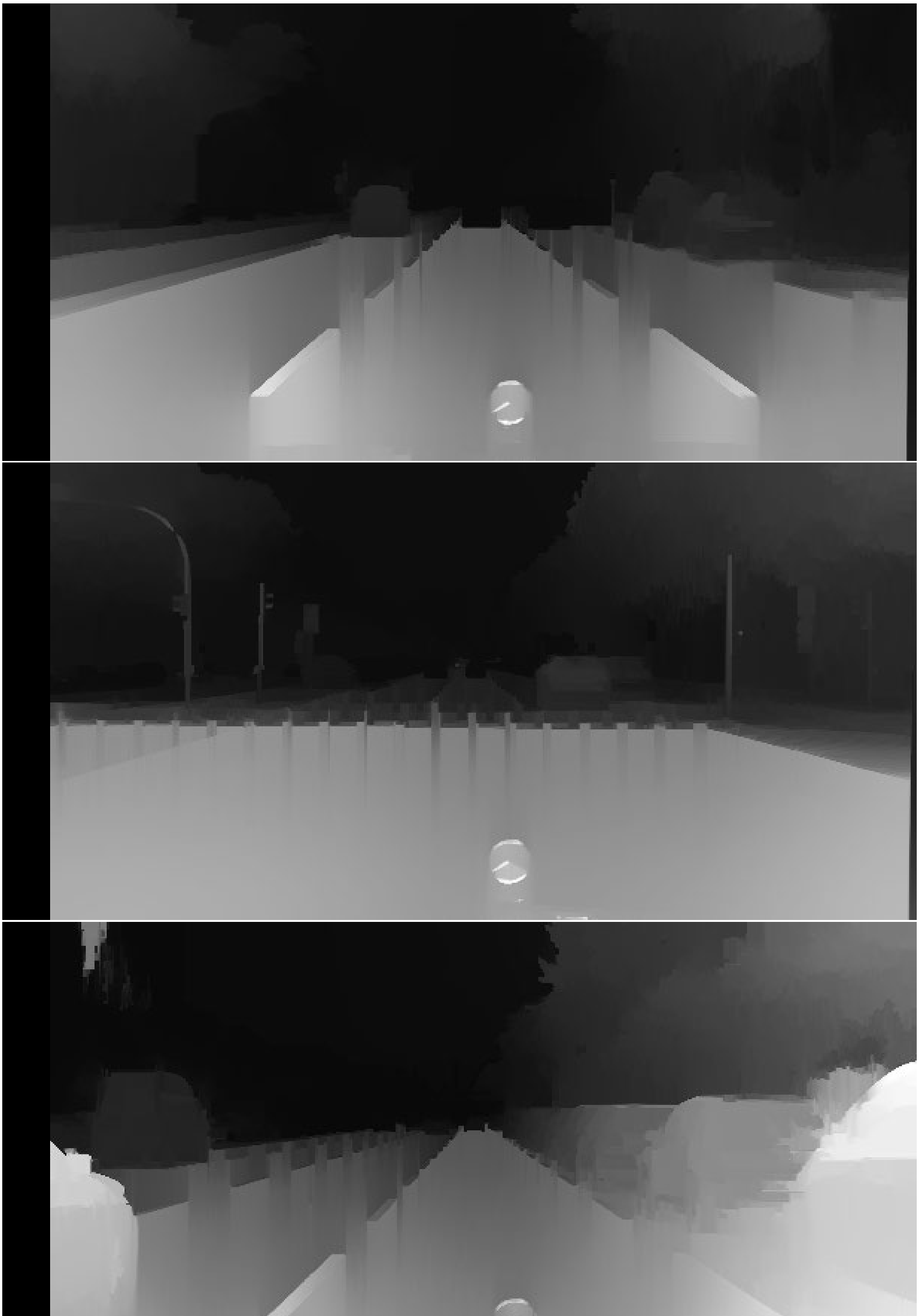


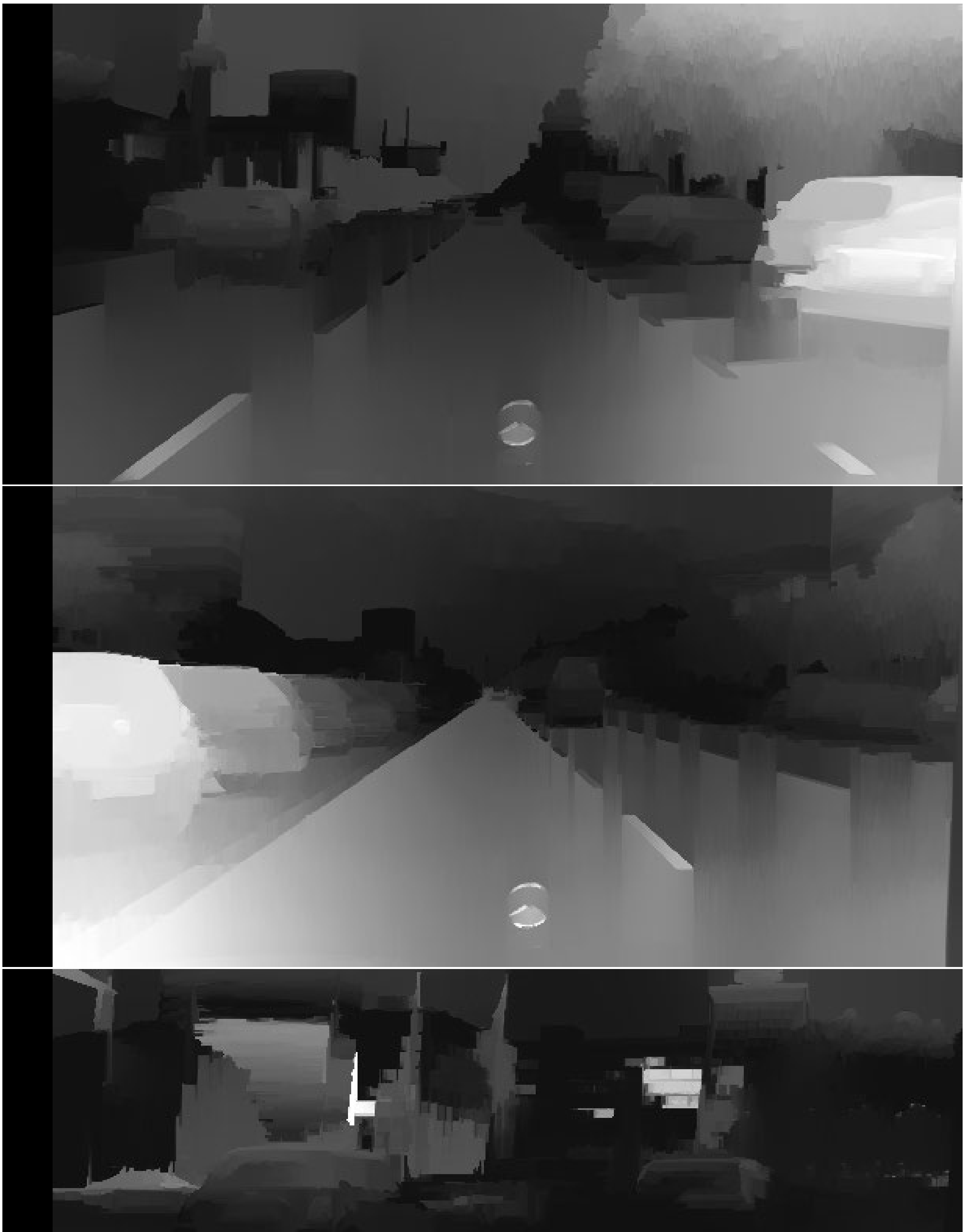


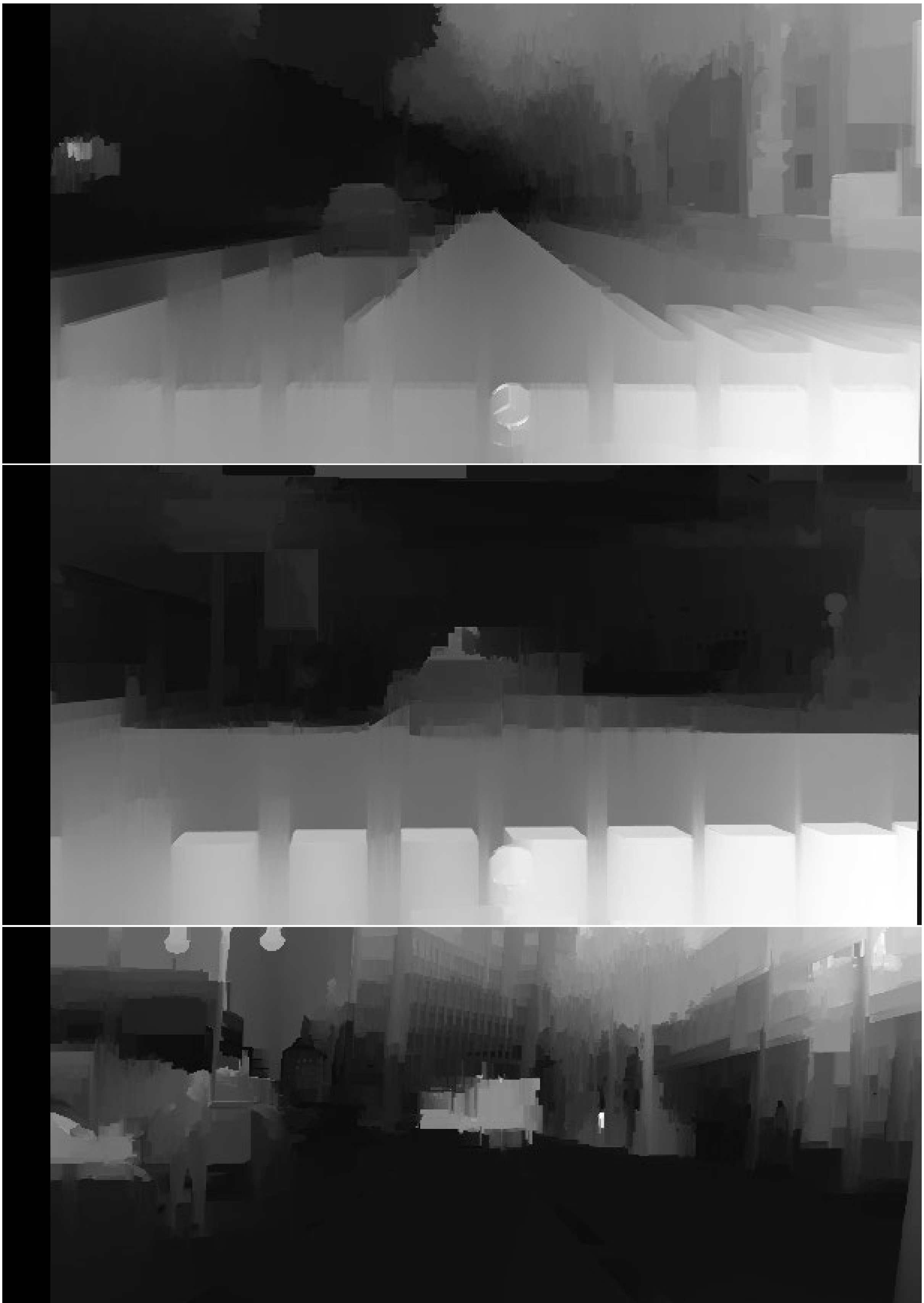


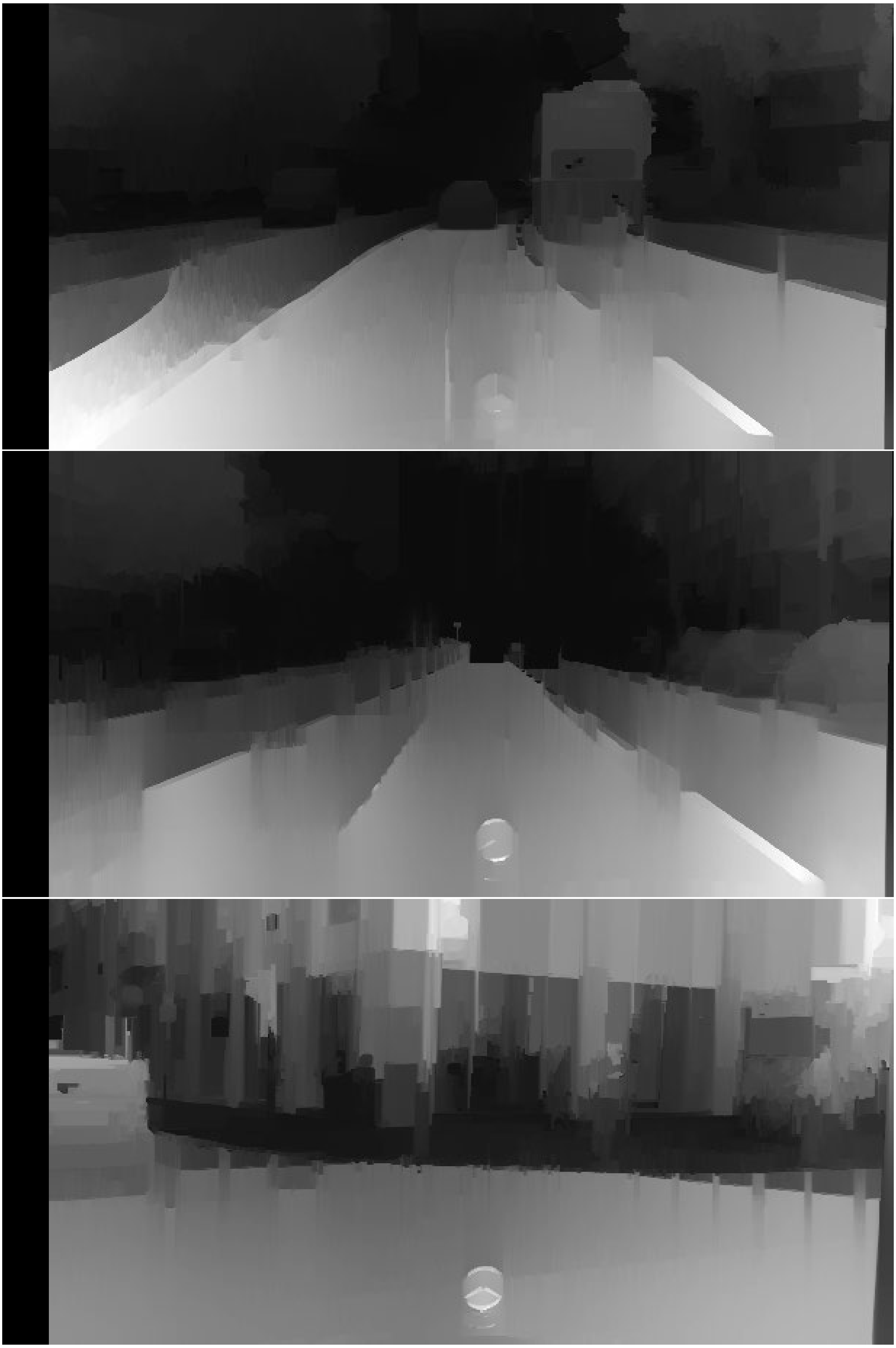


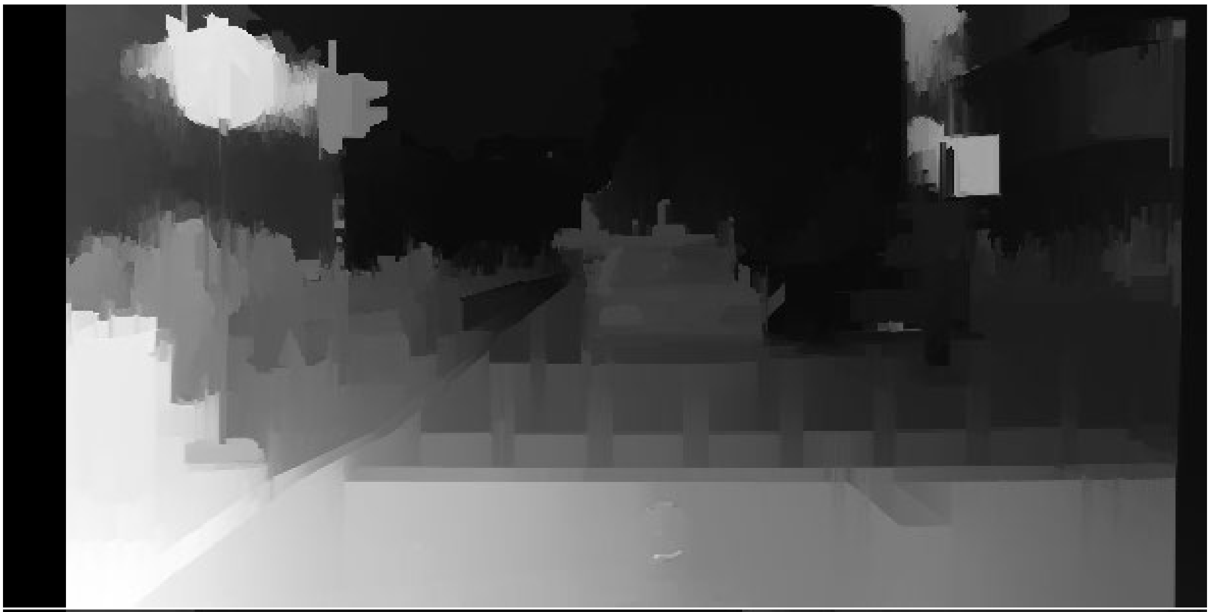


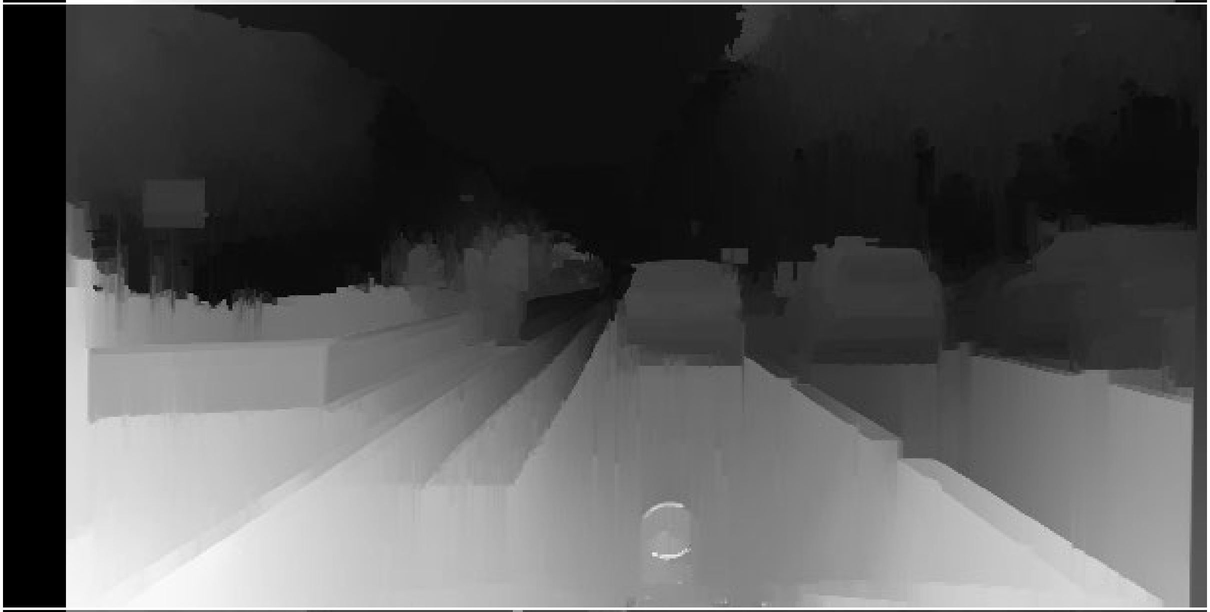


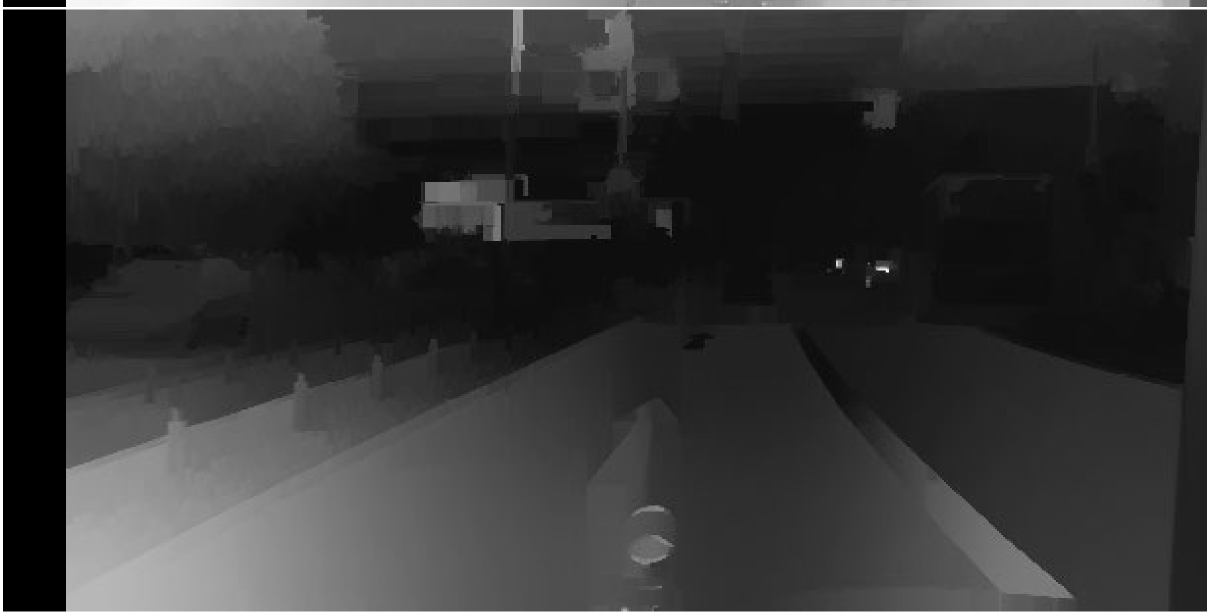


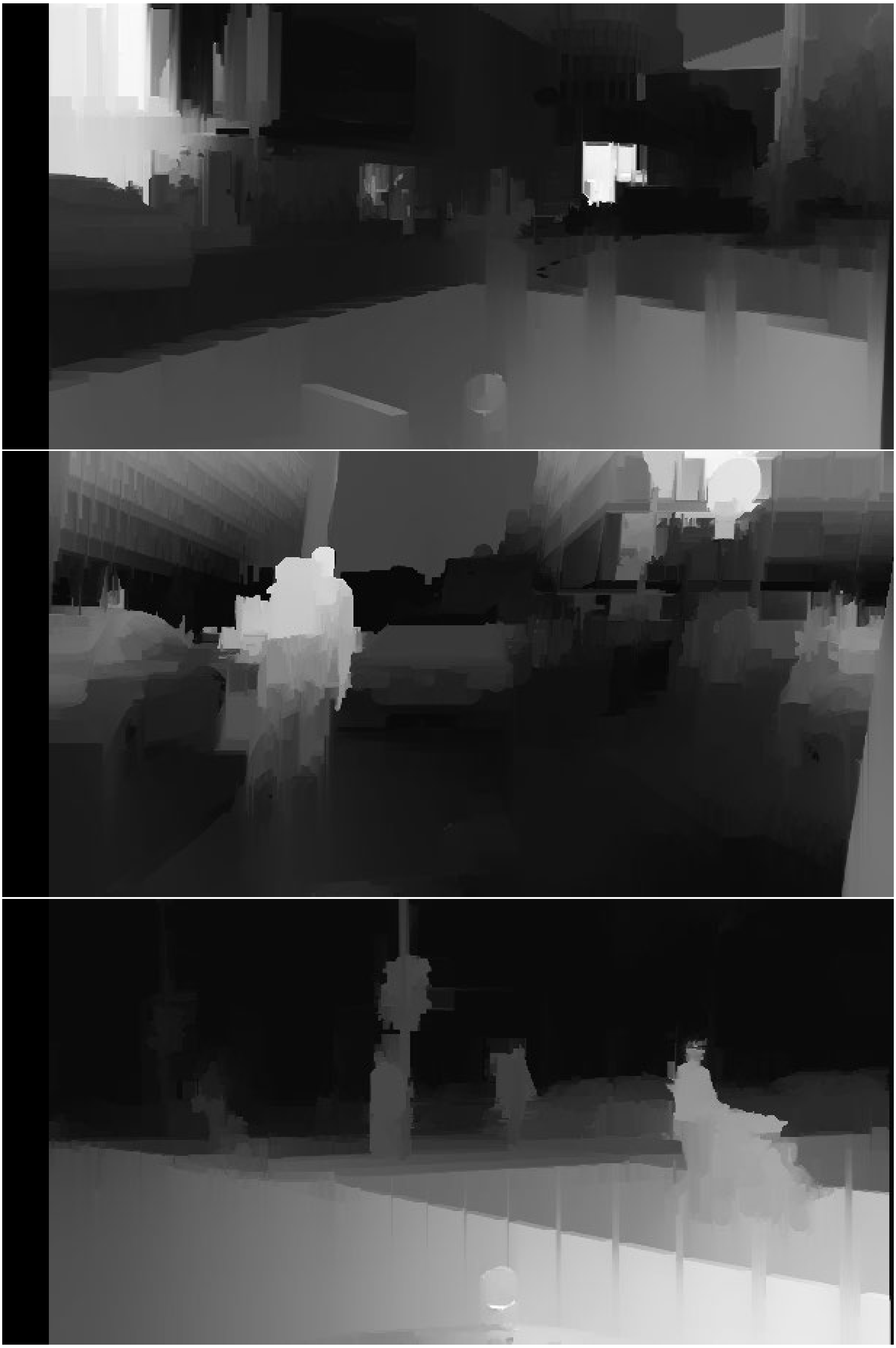


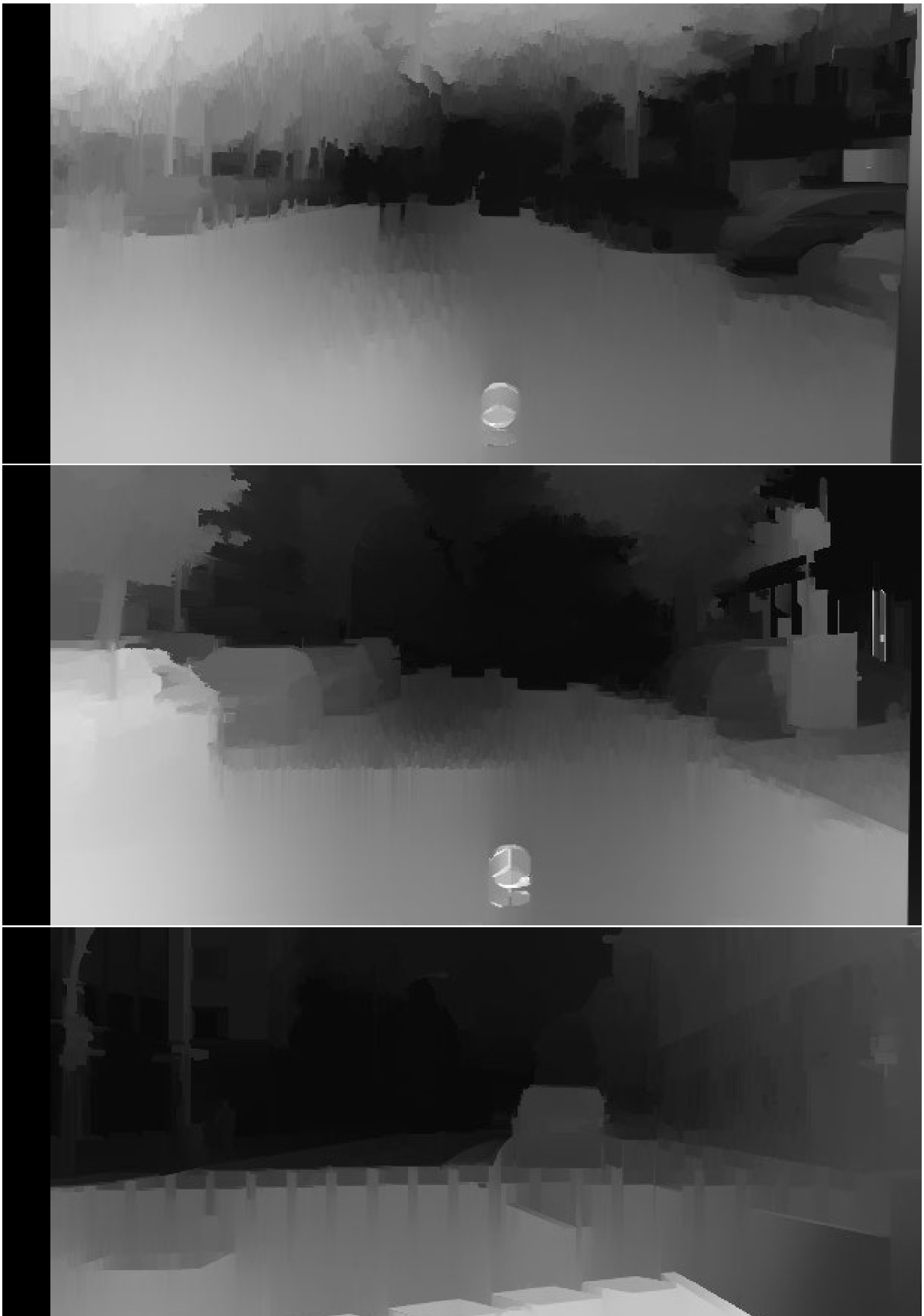


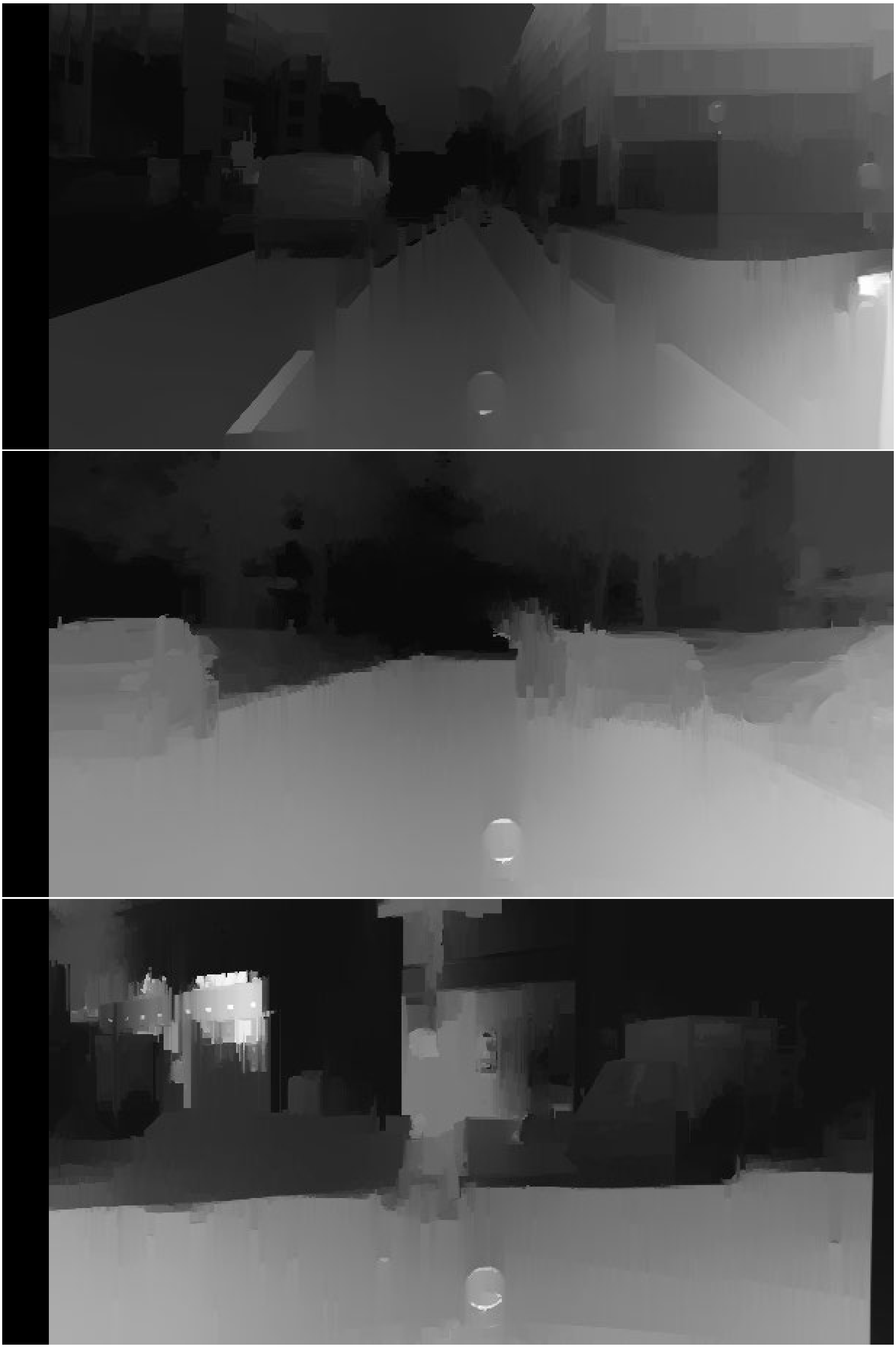


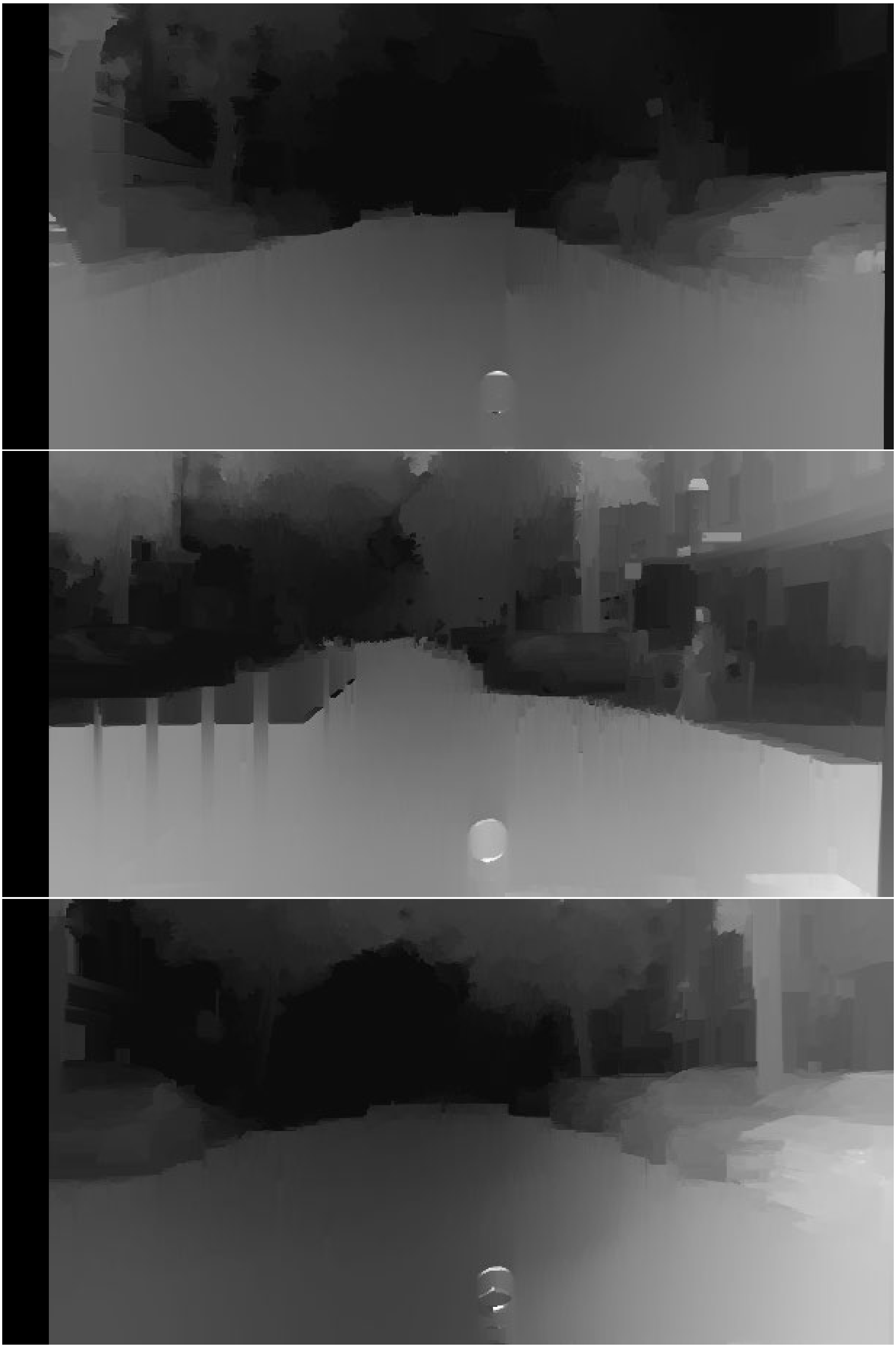


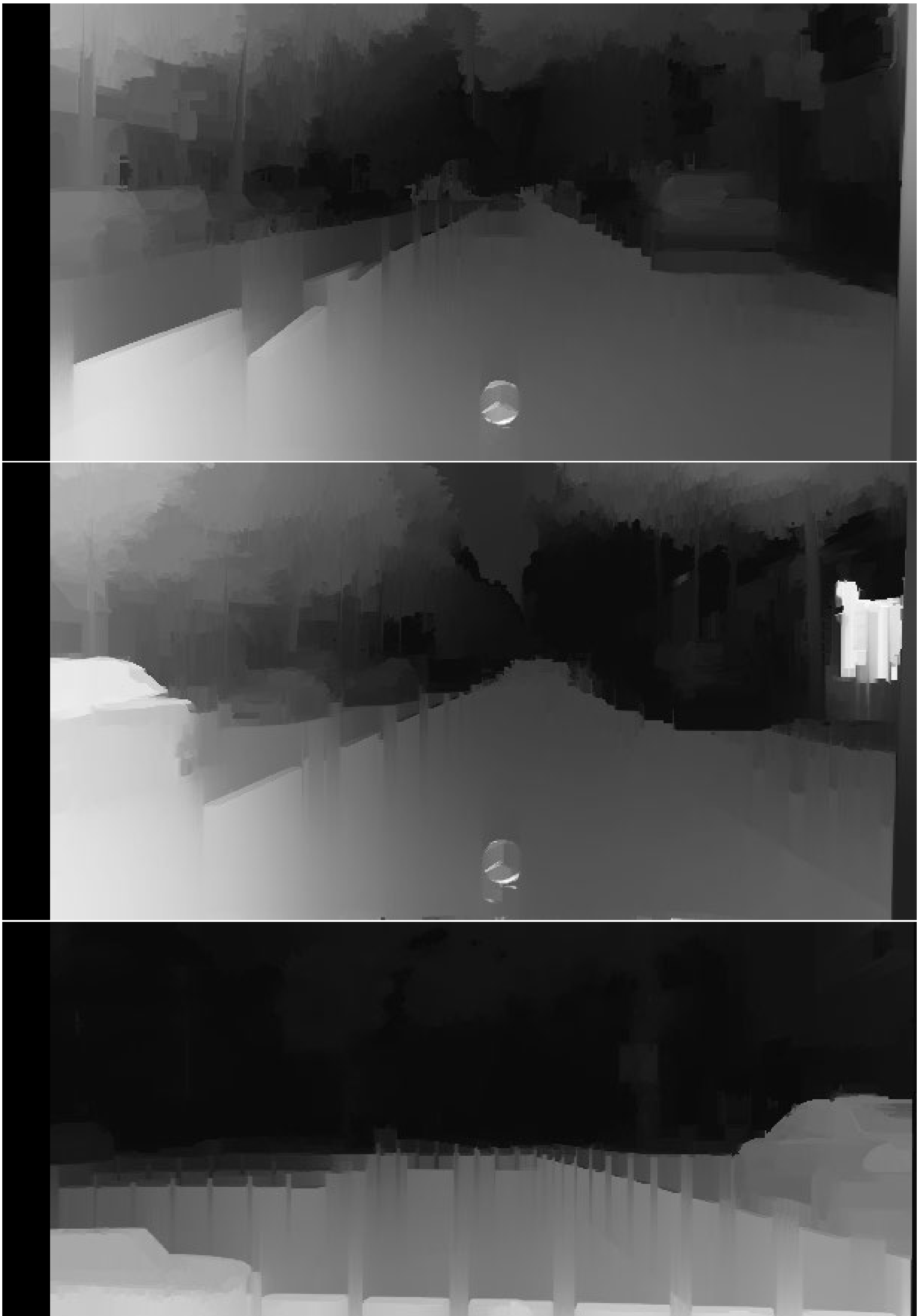


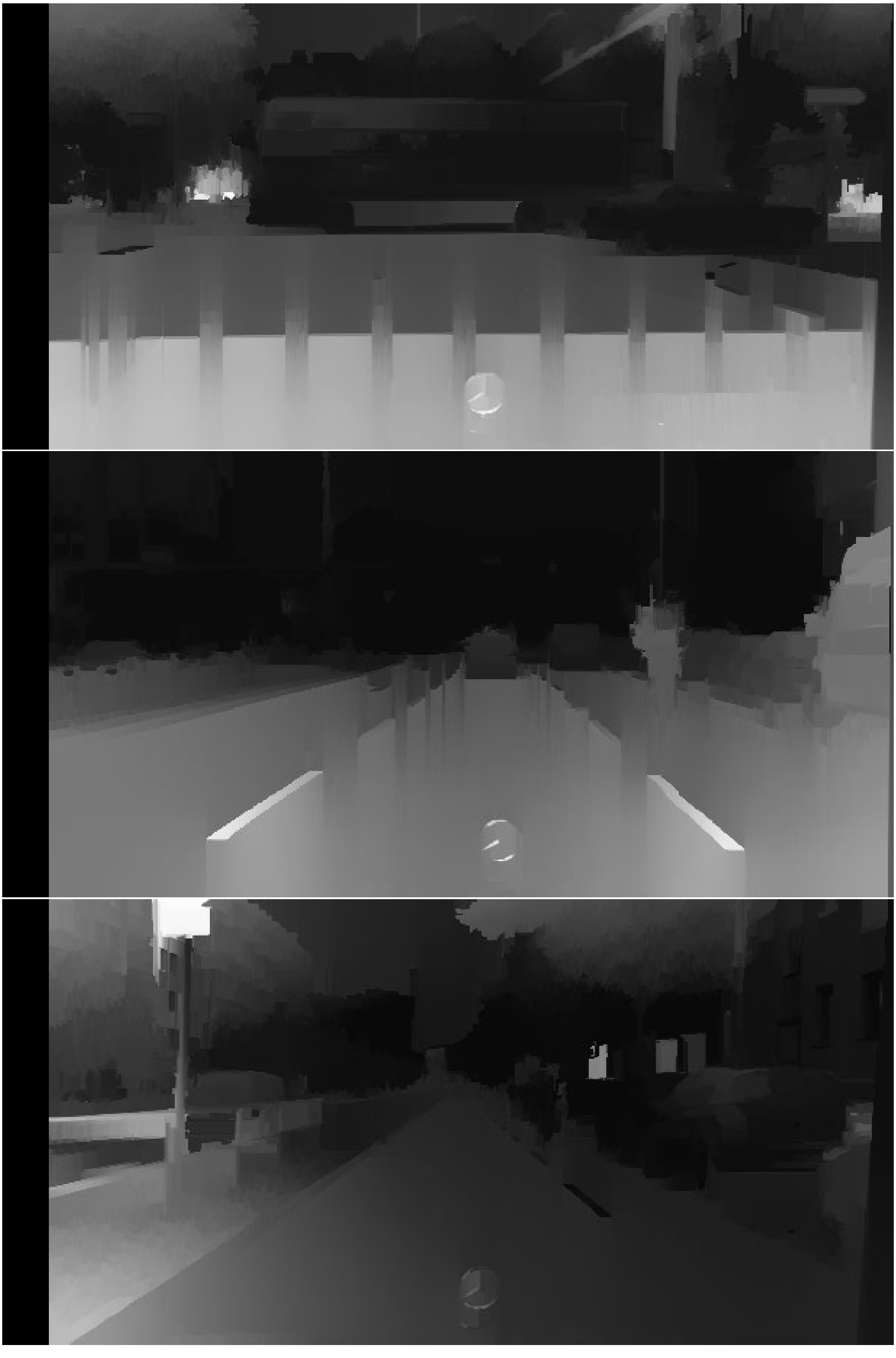


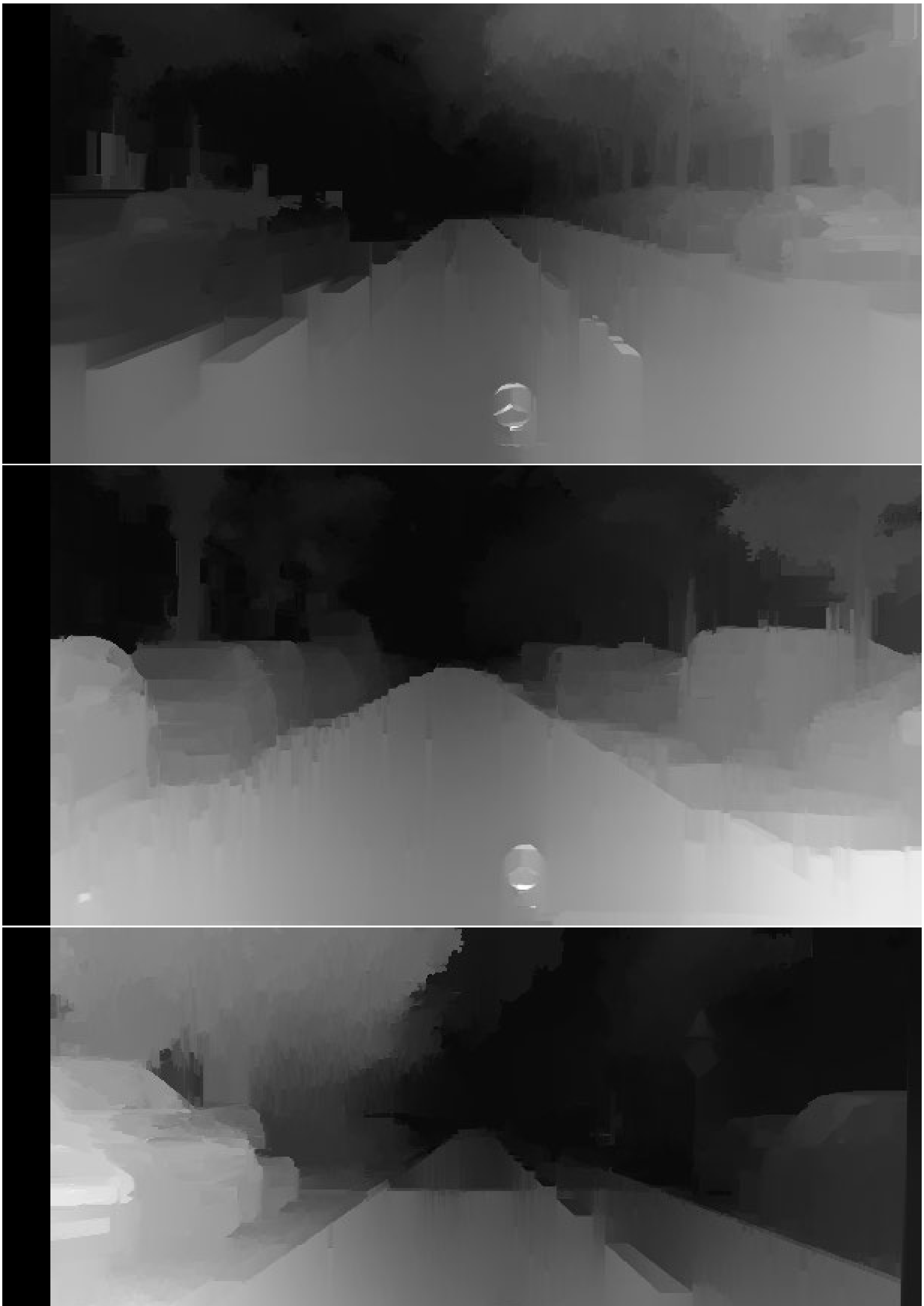


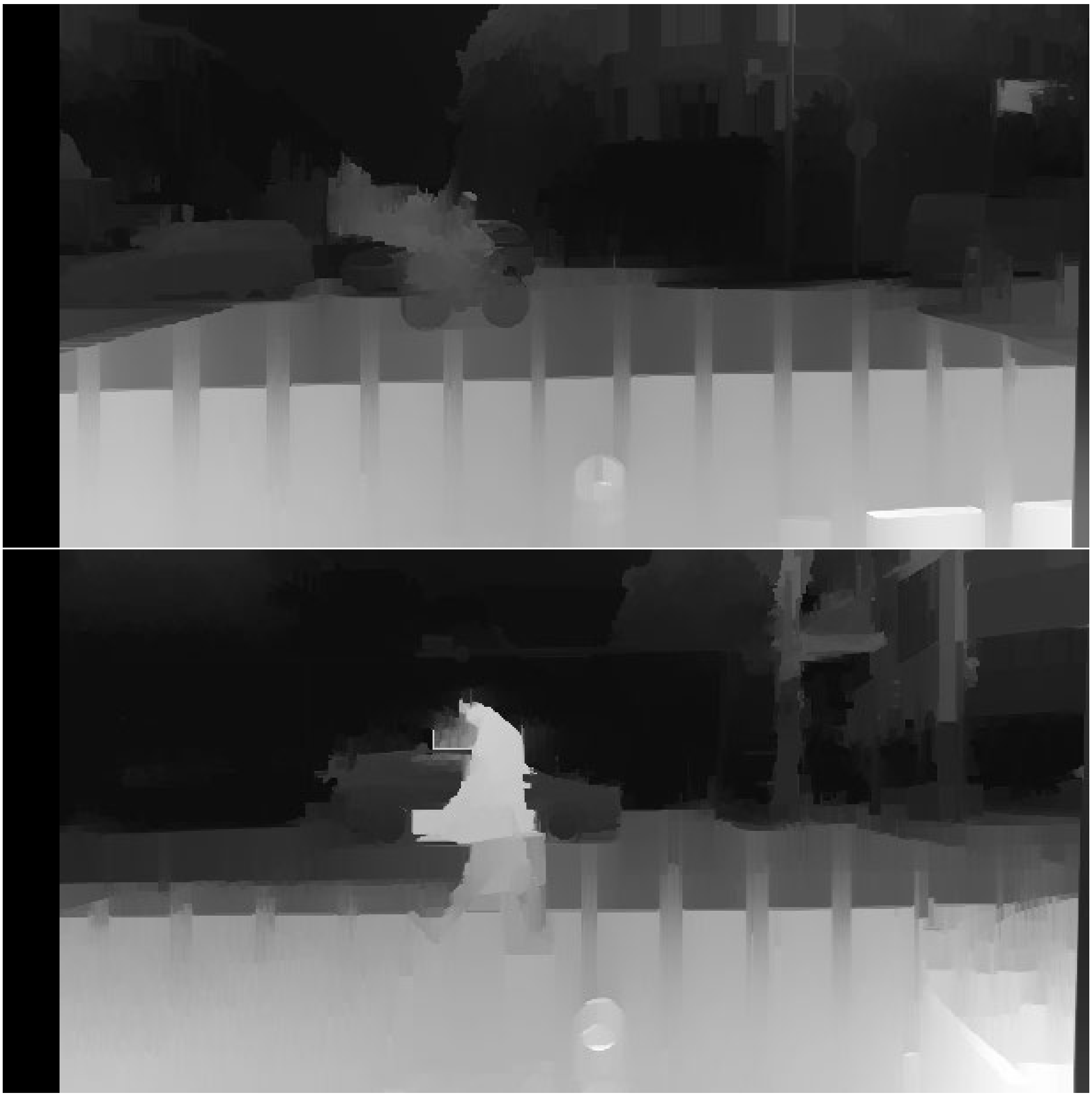


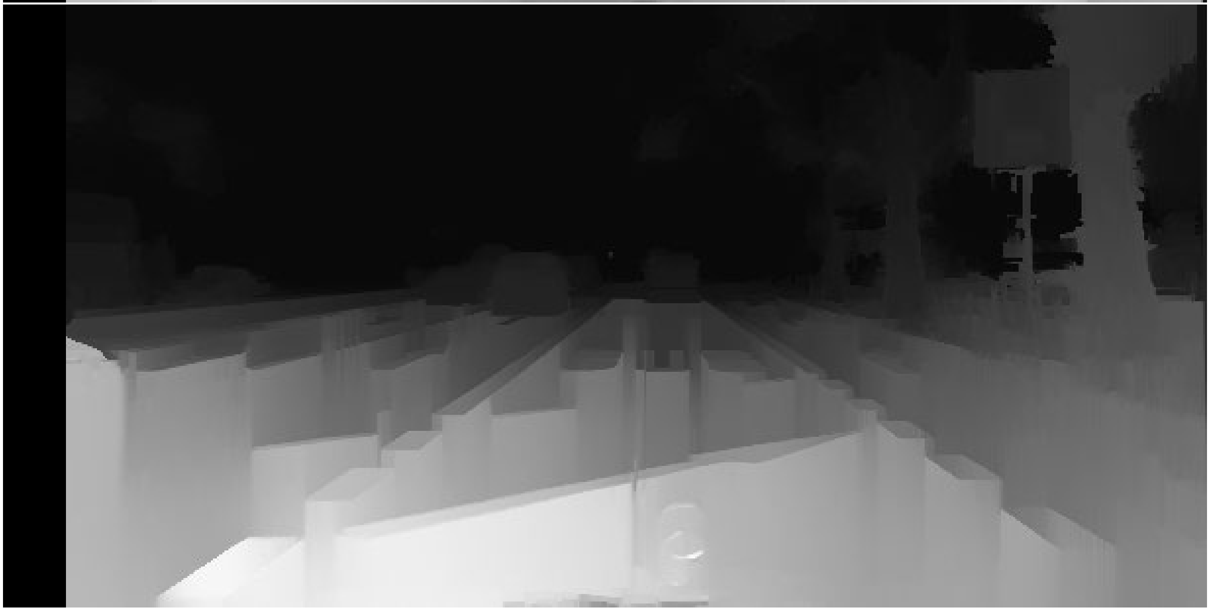












Sample 3rd Trial Disparity Maps SuperPoint: