SYSTEM DESIGN AND ANALYSIS FOR CREATING A 3D VIRTUAL STREET
SCENE FOR AUTONOMOUS VEHICLES USING GEOMETRIC PROXIES
FROM A SINGLE VIDEO CAMERA

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Timothy Wong

June 2019

© 2019
Timothy Wong
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE: System Design and Analysis for Creating a
3D Virtual Street Scene for Autonomous
Vehicles using Geometric Proxies from a
Single Video Camera

AUTHOR: Timothy Wong

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Zoe Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Bruce DeBruhl, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Jonathan Ventura, Ph.D.
Professor of Computer Science

ABSTRACT

System Design and Analysis for Creating a 3D Virtual Street Scene for Autonomous Vehicles using Geometric Proxies from a Single Video Camera

Timothy Wong

Self driving vehicles use a variety of sensors to understand the environment they are in. In order to do so, they must accurately measure the distances and positions of the objects around them. A common representation of the environment around the vehicle is a 3D point cloud, or a set of 3D data points which represent the positions of objects in the real world relative to the car. However, while accurate and useful, these point clouds require large amounts of memory compared to other representations such as light weight polygonal meshes. In addition, 3D point clouds can be difficult for a human to visually understand as the data points do not always form a naturally coherent object.

This paper introduces a system to lower the memory consumption needed for the graphical representation of an virtual street environment. At this time, the proposed system takes in as input a single front facing video. The system uses the video to retrieve still images of a scene which are then segmented to distinguish the different relevant objects, such as cars and stop signs. The system generates a corresponding virtual street scene and these key objects are visualized in the virtual world as low poly, or low resolution, models of the respective objects. This virtual 3D street environment is created to allow a remote operator to visualize the world that the car is traveling through. At this time, the virtual street includes geometric proxies for parallel parked cars in the form of light weight polygonal meshes. These meshes are predefined, taking up less memory than a point cloud, which can be costly to transmit from the remote vehicle and potentially difficult for a remote human operator to

understand. This paper contributes a design and analysis of an initial system for generating and placing these geometric proxies of parked cars in a virtual street environment from one input video. We discuss the limitations and measure the error for this system as well as reflect on future improvements.

ACKNOWLEDGMENTS

Thanks to:

- My mom, dad, and brother for always being by my side and constantly praying and supporting me.
- Sammy, Nate, and Zach for keeping up with my late nights and walking with me through it all.
- My wonderful professors at Cal Poly especially Dr. Zoë Wood for and guiding me and being patient with me through the thesis process.
- My great thesis group Amy and Kirsten (and Kole) who struggled with me through it all.
- The community of EPIC who are always for me there when I need it.
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

	Page
LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF ALGORITHMS	xiv
CHAPTER	
1 Introduction	1
1.1 Data Acquisition Practices	3
1.2 Point Cloud Difficulties	4
1.2.1 Data Visualization	4
1.2.2 Memory Consumption	5
1.3 Contribution	6
2 Background	8
2.1 Neural Networks	8
2.1.1 Supervised vs Unsupervised	9
2.1.2 Convolution Neural Networks	10
2.2 Semantic Segmentation	11
2.2.1 Mask-RCNN	12
2.3 Depth Estimation	14
2.3.1 Monodepth	15
2.3.2 LiDAR	18
2.4 Related Works	18
2.4.1 3D R-CNN	19

2.4.2	Monocular 3D Pose and Shape Estimation of Multiple People in Natural Scenes	20
2.4.3	Comma.AI	21
3	Implementation	23
3.1	Overview	23
3.2	Virtual Street Scene Background	24
3.2.1	Roads	25
3.2.2	Blocks	27
3.2.3	Buildings and Sidewalks	27
3.3	Positioning and Drawing of the Specific Objects in the Scene	28
3.3.1	Segmentation	29
3.3.2	Locating Position	30
3.3.2.1	Position Intermediate Representation	31
3.3.3	Determining Depth	33
3.3.4	Metadata Compilation	34
3.3.4.1	Depth Intermediate Representation	38
3.3.5	Cars and Stop Signs	38
4	Results and Validation	42
4.1	Data Gathering	42
4.2	Analysis	43
4.2.1	Visual Accuracy	43
4.2.2	Quantitative Accuracy	47
4.3	Memory Intensity	53
4.4	Time consumption	55
5	Conclusion	57

5.1	Future Work	57
5.1.1	Depth	57
5.1.1.1	Monocular Depth	57
5.1.1.2	Computer Stereo Vision	59
5.1.1.3	Photogrammetry Software	62
5.1.1.4	LiDAR	63
5.1.2	World Generation	64
5.1.2.1	Data Gathering Challenges and Tips	65
5.2	Conclusion	66
5.2.1	Vision Moving Forward	68
	BIBLIOGRAPHY	70

LIST OF TABLES

Table		Page
4.1	Table showing the distance intervals and their respective standard deviations and means	50
4.2	Distance differences based on intervals	52

LIST OF FIGURES

Figure	Page
1.1 (a) Example of Sparse Point Cloud, (b) Example of Dense Point Cloud [12]	4
1.2 Current Point Cloud Visualization [8]	5
1.3 Example Point Cloud of a Lecture Room [39]	6
2.1 Example of CNN [37]	11
2.2 Example Output of Mask R-CNN [23]	13
2.3 Example depth map generated using monodepth. [18]	16
2.4 The network would take in a left image as input and output a disparity map of the right image. Using that disparity map, the network would then combine it with a sample from the left image to create the right image. However, as the input of this trained network iteration would be a left image, Godard et al. wanted the network to produce a disparity map of the left image, not the right image. This was rectified by training the network to predict the disparity maps for both images by sampling from opposite input images. [18] . . .	17
2.5 3D R-CNN implementation results [27]	19
2.6 Visualization of Monocular 3D Pose and Shape Estimation Algorithm [46]	21
2.7 Comma AI in use [3]	22
3.1 Overview of World Generation System	24
3.2 View of fully generated world	26
3.3 Examples of roads with different number of lanes	27
3.4 View with blocks	28
3.5 View of fully generated world	29

3.6	Segmented image and it's different given coordinates. Origin is located at the top left of the image with the x coordinates increasing moving to the right and the y coordinates increasing moving down. The depth is gathered by looking up the center value found in the segmented image in the depth map array.	32
3.7	Comparison of Monodepth and Monodepth2	35
3.8	Actual depths vs predicted depths	37
3.9	Example of systems used	40
3.10	Example Final View	41
4.1	Example original test image (top) and modified test image with extraneous information blacked out (bottom).	44
4.2	Good scene representation as all objects match their respective positions in the original.	44
4.3	Good scene representation as all objects match their respective positions in the original.	45
4.4	Incorrect scene representation as there is supposed to be a car behind the pink car on the right.	45
4.5	Incorrect scene representation as the cars are all too close to the camera.	46
4.6	Graph showing the predicted distances against the actual distances.	47
4.7	Graph showing a five-number summary of the predicted distance difference error. The lines for each column shows in ascending order the minimum, first quartile, median, third quartile, and maximum values for the data in the 10 feet range.	48
4.8	Graph showing the ratio of samples to the amount of difference for each increment of 10 feet. The orange bars represent how many samples were in that 10 feet range while the blue line shows the average difference between the actual and predicted distances for cars in each respective distance interval. The yellow line shows the total average distance over all the prior intervals.	49
4.9	The textured and untextured meshes created using <i>Meshroom</i>	54

5.1	Two input images left and right images and the corresponding depth maps created. Both depth maps are of the left image, but StereoPhoto Maker uses both left and right images to compute it while monodepth2 only uses the left image.	61
5.2	The textured and untextured meshes created using <i>Meshroom</i>	63

LIST OF ALGORITHMS

3.1	Mask-RCNN Filtering	31
3.2	Position Intermediate Representation	33
3.3	Depth Intermediate Representation	38
3.4	Metadata Compilation	39

Chapter 1

INTRODUCTION

Since the late 1920s, people have sought to create autonomous, or self-driving, cars [1]. While early work proved to have little practical application or success, improvements to processing power leading to machine learning and artificial intelligence has once again reignited the push for autonomous cars [1]. Companies such as Google and Tesla have already begun to mobilize autonomous cars proving that the technology for autonomous cars is rapidly improving [11]. To measure this, the Society of Automotive Engineers (SEA) created a level system for how autonomous a vehicle is. This scale ranges from zero to five with zero having little autonomous features and five having fully autonomy and not requiring a driver [2]. Google's car Waymo operates at a level 4 driving automation meaning it does not always require a driver to be in the car and can drive in specific environments [24]. Tesla's car operates at a level 2 driving automation meaning that it requires a driver to be in the car, but still can drive on its own in certain scenarios [26]. However, these cars are not truly autonomous and still require a driver to be in the vehicle and contain steering wheels and pedals. As such, other companies, such as GM and Zoox, have begun to research level 5 autonomous vehicles, which do not require a driver but can drive on its own in any condition. These cars would not have any designated space for a driver nor have any method of driving the car manually as they would not contain steering wheels or pedals [22, 43].

These vehicles are very promising, but bring about a new set of problems to autonomous vehicles. One such problem, which this paper addresses, is that these autonomous vehicles require a representation of the world around them in order to

navigate. Understanding what an autonomous vehicle’s environment is important for all levels of autonomous vehicles but particularly for level 5 autonomous vehicles as there is no manual driver. One possibility coming out of driverless cars then is the employment of remote human operators in case of emergencies that the vehicle cannot navigate through. Even if a driverless car can drive well in most conditions, there may still be certain scenarios that would benefit from having human perspective and guidance in navigating the car. These remote operators would need to quickly understand the scene the vehicle is in in order to navigate the car remotely, making the environment representation very important. In addition, the representation may be used for debugging of what a car is doing with respect to the input information. Point clouds are currently the most common and accurate way autonomous vehicles represent the world around them, but these point clouds are very memory intensive and can be difficult to understand from a human perspective as the point clouds have no structure to them.

This paper introduces a system to lower the memory consumption needed for the graphical representation of an virtual street environment. As acquiring point clouds are an expensive and difficult process, this thesis aims to see how using a front facing video taken from any single camera can be used to represent the environment around it. As a result, the proposed system takes in as input a single front facing video of the current scene. The system uses the video to retrieve still images which are then segmented to distinguish the different relevant objects in the environment, such as cars and stop signs. The system then generates a corresponding virtual street scene and these key objects are visualized in the virtual world as low poly, or low resolution, models of the respective objects. This virtual 3D street environment is created to allow a remote operator to visualize the world that the car is traveling through. At this time, the virtual street includes geometric proxies for parallel parked cars in the form of light weight polygonal meshes. These meshes are predefined, taking up

less memory than a dense point cloud, which can be costly to transmit from the remote vehicle and potentially difficult for a remote human operator to understand. This paper contributes a design and analysis of an initial system for generating and placing these geometric proxies of parked cars in a virtual street environment from one input video. We discuss the limitations and measure the error for this system and reflect on future improvements.

1.1 Data Acquisition Practices

Currently, most autonomous cars drive around by first performing image segmentation of input video in order to build up a representation of the world around it. They also construct point clouds, or sets of 3D data points, of the environment to continue building up an accurate representation. Point clouds are normally acquired using a piece of equipment called Light Detection and Ranging (LiDAR) and takes two different forms - dense point clouds or sparse point clouds. As Figure 1.1 shows, there is quite a difference in the level of detail when it comes to dense and sparse point clouds. While dense point clouds are more useful in the sense that they have more data that a vehicle can use for building a representation of a scene, they are not easily obtained as high resolution LiDARs are very expensive and can only make dense point clouds at a short range [49]. On the other hand, sparse point clouds are more easily generated as any LiDAR can do so. While less detailed, sparse point clouds in conjunction with video still typically provide enough information for a vehicle to understand and navigate the world around it [47].

Figure 1.1: (a) Example of Sparse Point Cloud, (b) Example of Dense Point Cloud [12]

1.2 Point Cloud Difficulties

1.2.1 Data Visualization

One challenge with point clouds is their visualizations. While a sparse point cloud combined with video can provide enough information for an autonomous vehicle to navigate, they can often prove to be more difficult for humans to visualize as the data points do not always form a naturally coherent object [12]. Advances in LiDAR technology and algorithms have led to better visualizations of sparse point clouds as shown in Figure 1.2 with color shading and wave-like scans, which help give a sense of depth and structure of the environment. However, these visualizations can still be difficult to parse and understand from a human standpoint as each point is independent and does not communicate with other points to form a coherent object. These points then can form arbitrary shapes that can make it hard to distinguish what an object is.

Figure 1.2: Current Point Cloud Visualization [8]

1.2.2 Memory Consumption

Another challenge with point clouds is that they take up large amounts of memory. For example, Figure 1.3 shows a dense point cloud consisting of 44,574,647 points with a memory size of 1.6 GB [39]. A sparse point cloud may contain only a tenth of that amount of points, but that still means it would take 160 MB of memory [34]. This brings about the problem of data transmission. As driverless vehicles may have a remote operator to drive or debug data from the vehicle, the point cloud would have to be transferred from the car to the location of the operator. In addition, as the vehicle will be moving while in operation, data transfer rates will be very limited and slow. One metric to compare this with is with mobile networks. Currently, Verizon has the fastest mobile data upload rate with a maximum speed of 65.2 Mbps and an average speed of 19.0 Mbps [40]. If transferring a sparse point cloud at 160MB like the example above, the upload would take 20 seconds at 65.2 Mbps and 1 minute 10 seconds at 19 Mbps. This means that there will be major lag between the vehicle

Figure 1.3: Example Point Cloud of a Lecture Room [39]

and the operator as the operator would want to see what the vehicle is seeing in real time.

1.3 Contribution

A solution to these types of problems would be to take a step back from point clouds and instead focus on representing a scene in a different way. As such, rather than relying on a point cloud for the environment representation, the main contribution of this work will be proposing a system for reconstructing generalized scenes in order to improve visualization and alleviate memory consumption. These scenes will only contain parked cars in order to focus the scope of this project. This system will take in as input a single camera video source and will use still images from it to learn where to position each object in the scene. The system will then generate a corresponding virtual street scene and visualize each object in the virtual world as low

poly, or low resolution, model. These meshes are predefined, taking up less memory than a dense point cloud, allowing for easier and quicker transmission of data.

An inspiration for using a single camera is that some autonomous vehicle companies are opting to use cameras rather than LiDAR for autonomous driving. For example, Tesla has publicly refused to use LiDAR as they are expensive and unnecessary [7]. Other companies such as Comma.AI are also working to see if using only smart phones could be used for autonomous driving rather than having specific cameras made for autonomous driving [3]. As a result, we constrain our system to single camera video to test and experiment how effective it can be.

This project will also explore the different limitations to this system as well as what different options and steps can be taken to improve this project.

Chapter 2

BACKGROUND

This project combines different computer vision and machine learning concepts to produce the virtual street representation. We provide background on semantic segmentation and depth estimation with focuses on neural networks and specific the algorithms used for each concept. In addition, we look at how some of these concepts are also used in other works and how they compare and contrast to this project.

2.1 Neural Networks

A difficulty of semantic segmentation is that a computer must learn how to classify what the different objects in an image are. While humans can easily identify what objects are, computers cannot do the same with as much ease. Computers lack the prior knowledge that humans have in recognizing objects, as humans begin classifying objects as soon as they are born and build up a large database of what objects are [4]. As a result, computer must simulate/emulate the learning process in order to begin to classify objects in a process called machine learning.

One way machine learning is done is via neural networks, which are computer networks with a structure emulating neurons in a human brain. Most neural networks act like a function where they take in some input and produce an output. To learn how to produce the correct and desired output, the neural networks undergo a process called training. In training, the network is given large amounts of data and the desired outputs for that data. Through essentially trial and error, the networks learn how the input data relates to the desired output. The relations are then represented via

weights, which are stored in nodes. These nodes take in data, perform an operation on the data based on the weights, and then send the result to other nodes which proceed to do the same. The weights change as each iteration of data is passed through it until the weights properly represent the relation between the input and output data, simulating learning. This process is done in layers of nodes such that each node in one layer only communicates with nodes in the next layer. After the data goes through the all layers of the network, it is typically outputted to a vector that contains the probabilities of what an object is [48].

To see the success of the neural network, the networks are tested with validation and test data. These data sets are normally created before a network is trained where the entire dataset is split into training, validation, and test datasets. Validation data is used while training to indicate how well a network is doing while training and help the network adjust to perform more accurately. Testing data, on the other hand, is used solely for showing how a network performs after training. Both these sets accomplish their respective purposes by comparing the network output with a ground truth output with what is called a loss function. Loss is defined as the error or cost of the network. There are a vast number of different loss functions that can be used, but generally they describe in one way or another how close the network output is to the actual ground truth output. The smaller the output of a loss function, the better the performance of the network. Common loss functions include regression loss, binary classification loss, and multi-class classification loss [35].

2.1.1 Supervised vs Unsupervised

One of the types of learning used with machine learning is supervised learning. Supervised learning is performed by a neural network that takes in as input training data as well as ground truth. Having ground truth means that the network has some

prior knowledge of what results should come from the input training data, hence the term supervised learning. The goal of supervised learning is to learn and form a function that best approximates the relationship between the input and the ground truth. This allows the network to be able to take in some input without ground truth and still output a result that would be close to what the ground truth of the input would have been. Supervised learning is very useful for classification or categorization as it excels at matching one thing to another [41].

The other main type of learning is unsupervised learning. Unsupervised learning, as its name suggests, is the opposite of supervised learning in that it does not take in any ground truth data. As a result, networks operating with this type of learning take in some input and try to infer the natural structure of that input. This is especially useful if the structure of the data is unknown as unsupervised learning can infer trends in the data and help give some structure to the data [41].

2.1.2 Convolution Neural Networks

Convolutional neural networks (CNN) are a specific type of neural networks focused on taking images as input. The major difference between CNNs and general neural networks are that CNNs have one or more layers of convolution units. These units take in input from multiple units in the previous layer and combines them together to form a single output to be passed onto the next layer, reducing the amount of data passed through each layer. This is extremely useful for images as images contain massive amounts of data that would take a very long time to process if the data was not reduced. In addition, data in images are generally related to the data around it, allowing convolution units to gather and consider data from a related "neighborhood" of data [45].

One downside to CNNs are that they still take in so much data that they take a while to run. As a result, there are many variations of CNNs for specific purposes. Different CNNs use different sizes and types of layers in order to make training more efficient, as all the input data is not always useful for the CNNs purpose. Different types of layers narrow down the data with some emphasizing more important data while removing less important data and others expanding the data passed to it to analyze it more specifically. Spending more time analyzing data may improve the accuracies of the classifications but analyzing too much data exponentially increases the time it takes to train. Efficiencies of different CNNs depend on the ordering and types of layers in order to balance the amount of time spent and accuracy gained [20].

Figure 2.1: Example of CNN [37]

2.2 Semantic Segmentation

In computer vision, segmentation is the splitting of an image into different parts without any specification or understanding of what parts are split. Common ap-

proaches to this include finding edges in an image, creating boundaries by looking at the changes in color values across an image, or attempting to extract global impressions or patterns of the image [25]. While useful in certain cases, a more effective use of segmentation is to combine it with semantics. This process, known as semantic segmentation, is the splitting of an image into different parts and classifying each part into a pre-determined class. With images or videos, this correlates to classifying and labeling each pixel as a certain class, e.g. a car, a pedestrian, or a building.

2.2.1 Mask-RCNN

A useful way of segmenting images is by creating a bounding box around each object in the image. However, a problem that CNNs face with this is that the output vector of CNNs are constant, while the number of occurrences of objects in the image are not fixed and thus require a variable output vector. As a result, a family of CNNs called Region-based Convolutional Neural Networks (R-CNNs) were created to solve this problem. This family consists of four main iterations of CNNs, R-CNN, Fast R-CNN, Faster R-CNN, and Mask R-CNN. To combat the problem above, R-CNN first began by using selective search to extract 2000 regions from the image called region proposals. These regions are then sent into a CNN to extract features, which are used in another machine learning algorithm to determine where the corners of a box around those features are [16]. This concept was extended by Fast R-CNN which gave the original CNN the whole original image as input instead of 2000 regions to get a feature map. Selective search was used again to get the region proposals and was then inputted to another CNN that classified the region proposals and created boxes around the features [15]. Faster R-CNN improved Fast R-CNN by removing the selective search and replacing it with a neural network that predicted the region proposals [32].

Figure 2.2: Example Output of Mask R-CNN [23]

A problem with the first three iterations of R-CNNs was that they only produced bounding boxes. While suitable in some cases, this meant that the detection of objects in the image did not have any pose or representation of shapes. He *et al.* solved this problem by making Faster R-CNN perform pixel-based classification in addition to region based classification with Mask R-CNN [23]. Mask R-CNN branched off of Faster R-CNN after the features were defined by using the features to create an output matrix of 1s and 0s that represented what was an object and what was not respectively rather than outputting just the corners of a bounding box. This allowed Mask R-CNN to be more visually accurate and fine-tuned for semantic segmentation. In addition, He *et al.* used a different feature extractor as well as multitask training to improve the speed and accuracy of the segmentation.

2.3 Depth Estimation

Depth estimation is another big focus in the computer vision world today. In this particular case, depth estimation will refer to where a program tries to output a realistic depth value from an image or video. This is quite a hard challenge as even humans have a very hard time accurately estimating depth at almost any scale without the assistance of a tool.

There are currently two different approaches to depth estimation - using stereo or monocular images. The first is a stereo system. In the scope of this thesis, the style of stereo depth estimation discussed will be techniques using a single camera. These networks are usually trained by taking as input a series of forward movement images as well as ground truth depth maps of these images. They look at images two or three at a time to match patches of the images together and determine how far those patches in the images moved between them. This distance of movement between the images is called disparity. These outputs are then compared to the actual distances of the objects and allow the networks to refine its outputs to be more accurate [38, 44]. While this method is quite successful as any change in an image allows for a good estimation of depth, it is not always practical as two or three images are needed as input for the network to output a depth map. This causes problems as it is not possible to get a frame of a scene in the future and so distances will always "lag" behind as the depth map created will always be for the previous frame.

The other approach to depth estimation is with a monocular (single image) system. The first big caveat with monocular depth estimation in general is that it is an ill-posed problem as there is no solution to get an exact depth from a single image. The reason for this is that a single image is two dimensional meaning it does not include depth. As a result, any single 2D image can be created from an infinite

amount of different 3D spaces meaning that any object in a single 2D image has an infinite amount of different depths it could be at [5]. The way most monocular depth estimation networks work is that they are trained with series of three images and possibly ground truth data. The images consist of a current frame, the frame before the current frame, and the frame after the current frame. Using these images, the network detects the differences between the three images with the current frame being the focus. Doing this allows the network to pick up on depth cues about the current frame and make depth predictions from that. The resulting network can take in a single image which represents the current frame and look at its depth cues to make a prediction [5].

2.3.1 Monodepth

Monodepth is an implementation of an unsupervised monocular depth estimation network by Godard et al. titled monodepth [18]. As with any other depth estimation network the goal of monodepth is to learn a function that can predict the per-pixel scene depth of an image. While many may try to approach this problem with a supervised outlook, Godard et al. argue that gathering the depth data for each image of training is impractical as it is expensive to do accurately and even the higher end hardware such as LiDAR can be imprecise when there is movement and/or reflective materials. As a result, they opted to view this problem from an unsupervised learning point of view. Figure 2.3 shows an example result.

With the absence of ground truth depth of training images, Godard et al. took a different approach to training their model. Instead of matching images with depth data, they instead aimed for pose depth estimation as an image reconstruction problem during training [18]. Their idea was to train a network with binocular stereo images (left and right images) and reconstruct the right image from the left. Figure

Figure 2.3: Example depth map generated using monodepth. [18]

2.4 explains the process they used to do this. The trained network would then be able to create both right and left disparity maps from just a single left image. From the disparity map, the depth can be inferred through extra calculations based on the intrinsics of the camera that took the photo.

Monodepth is trained with three loss functions created by Godard et al. These loss functions are appearance matching loss, disparity smoothness loss, and left-right disparity consistency loss. Appearance matching loss looks at the difference between the output image and the input image as they should be the same. Disparity smoothness loss looks to see if the output disparities are locally smooth and adds error the less locally smooth they are. Finally, left-right disparity consistency loss looks at the difference between the left and right disparity maps as they should be the same as well.

A second iteration of monodepth titled monodepth2 was released earlier this year which improved on monodepth by taking as input larger images. This allows the model to gather more details from the images and thus have more information to create more accurate depth maps. In addition, it jointly trained off of both stereo and monocular images rather than on just stereo images, which proved to provide a better result than the original monodepth algorithm.

Figure 2.4: The network would take in a left image as input and output a disparity map of the right image. Using that disparity map, the network would then combine it with a sample from the left image to create the right image. However, as the input of this trained network iteration would be a left image, Godard et al. wanted the network to produce a disparity map of the left image, not the right image. This was rectified by training the network to predict the disparity maps for both images by sampling from opposite input images. [18]

2.3.2 LiDAR

One of the most popular and accurate ways of acquiring point clouds is through LiDAR (Light Detection and Ranging). LiDAR works by sending out a pulsed laser and using the feedback from the laser bouncing back to record distances [17]. LiDAR gets its accuracy from the fact that it calculates distance using the speed of light and the laser feedback. As the speed of light is constant, LiDAR does not suffer from many environmental factors and relies only on the laser feedback. While different LiDARs operate differently, most LiDARs can send around 160,000 pulses of light from the laser per second allowing them to get very accurate readings of where points in space are [17]. There are two common types of LiDARs in use today - 2D LiDAR and 3D LiDAR. As the names suggest, 2D LiDAR devices produce a 2D point map while 3D LiDAR devices produce a 3D point cloud.

The major use of LiDAR in industry is for accurate distance measurements. Point clouds contain points that have an x, y, and z coordinate. Using those three coordinates, a program or user can calculate how far away a point in space is from the LiDAR in a 3D space. As a result with respect to cars, LiDAR devices are normally mounted on top of the car in order for the distances calculated from the LiDAR to be the distance from the car.

2.4 Related Works

With machine learning combined with computer graphics increasing in popularity and success, there has been an increasing number of works on related to building up virtual worlds of real life scenes. We take a look at some different methods and implementations of creating graphical representations of scenes in this section.

Figure 2.5: 3D R-CNN implementation results [27]

2.4.1 3D R-CNN

The framework of R-CNNs have continued to be a very prominent and effective model for segmenting images quickly. There have been four commonly known and accepted R-CNNs which include R-CNN (2013), Fast R-CNN (2015), Faster R-CNN (2015), and Mask R-CNN (2017) [16, 15, 32, 23]. Each of these iterations of R-CNNs have built up on each other with Mask R-CNN being able to segment, create bounding boxes, and create accurate masks for instances in an image. However, none of these algorithms estimate the 3D shape or position of the instances in the image. Kundu et al. attempts to do this with their 3D R-CNN algorithm [27].

3D R-CNN builds off of the previous versions of R-CNN and transfer that methodology from 2D to 3D. Kundu et al. build much of their model from the ground up as there was not much support from the previous iterations of R-CNN for 3D analysis. They focus solely on cars in their implementation. Their method starts by using ResNet-50-C4, a feature extractor, to grab features and feed that into their variation of a R-CNN which gives them information about the bounding boxes, the centers, the shapes, and the poses of the cars found in the scene [27]. This information then

is used in a method called render-and-compare loss to determine how far away the object is from the camera. This method works by rendering the cars into the scene using the center point, shape, and pose and comparing the bounding box and mask that car creates in 3D with its 2D counterpart. If they do not match, the algorithm shifts the car further away in the 3D scene and renders it again until both bounding boxes and masks match. They were able to expand this algorithm to human pose and shape as well showing the versatility of 3D-RCNN.

This method achieved very good results, having very realistic 3D masks for the scenes as seen in Figure 2.5. The main drawback for this method is the amount of computation needed for all the different networks used as well as the amount of rendering needed for their render-and-compare loss function. Without access to a powerful GPU or TPU, the training of their implementation can require a large amount of time and energy. In addition, while their render-and-compare loss is very efficient for objects with a well defined structure like cars and humans, they may struggle with objects that vary in structure such as landmarks or buildings as it is difficult to train a network on all the different structures.

2.4.2 Monocular 3D Pose and Shape Estimation of Multiple People in Natural Scenes

Zanfir et al. also try to obtain the pose and shape of instances in a scene to create a 3D representation of the scene [46]. In their implementation, they focus solely on humans in order to have a more focused scope. The algorithm they devise first sends their scene into a network which produces an output of person detection, body part labeling, and 3D pose initialization. That information is then send into a another network to determine the actual pose and shape. This is done per person in the scene and then sent to another model to piece all the people together and assign them to

Figure 2.6: Visualization of Monocular 3D Pose and Shape Estimation Algorithm [46]

correct positions.

This algorithm also works quite well and is able to determine accurate pose and shape even when people are occluded by other objects or people in the scene or interacting with others. This is accomplished as the network learns how body parts interact and how they should move. A sample of this is shown in Figure 2.6.

The main drawback for this method is that the data the model is trained is highly reliant on the attributes of people such as how body parts move and interact with objects. While it works extremely well with people, it cannot be directly transferred to be used with other objects. Still, their method provides some insight into different methods such as using multiple network models that focus on specific attributes as well as good model bases.

2.4.3 Comma.AI

Comma.AI is a company that has been aiming to add autonomous driving to existing cars that are not built or have any autonomous functions inherent to them. They aim to do this by using a smartphone to track and understand the environment the car is in. An example of their application in use is shown in Figure 2.7. Comma.AI is able to do this by having a user plug in a device into their car that allows the smartphone to connect to the car and control different aspects of driving the car including turning the steering wheel and pressing the gas and brake pedals [10].

Figure 2.7: Comma AI in use [3]

Their technology is a crowdsourced artificial intelligence-based advanced driver assist meaning that users can submit data to their platform so that the AI can train and learn how better to act in different situations [21].

The technology for this provides level 2 automation and as such currently cannot change lanes or recognize red lights or stop signs. However, it is able to provide adaptive cruise control which keeps the car a certain distance away from the car in front of it. It can also work under stop and go traffic as it will come to a complete halt if the car in front of it stops and begin to drive again after the front car starts moving [21].

Chapter 3

IMPLEMENTATION

This thesis proposes a system for the creation of a virtual street view of a given scene from a single camera. This system will take in video and gather still frames from it. From the still frames, it will identify the different cars and stop signs in the scene and place low poly geometric proxies in their respective places in the virtual scene. These objects will then be placed in a generic static environment. This virtual environment allows autonomous vehicles to represent the environmental data around them in a less memory intensive method compared to point clouds while also displaying the data in a more visually coherent manner. This helps address future problems where point clouds can be costly to transmit from remote vehicles and potentially difficult for a human to understand.

3.1 Overview

The implementation is split into two parts. The first part involves creating a 3D scene background through our virtual street scene application. For this project, the background scene is meant to provide context for the cars and stop signs we are specifically placing into the scene rather than to be a focus of what we want to achieve. As a result, this project does not represent roads and buildings in accordance to the input scene. Instead, the background street scene is drawn with generic objects. These generic objects consist of pre-positioned roads and buildings in order to help provide a general context for the scene and our specific objects, or objects that have a specifically determined position rather than a generic predefined position. The

Figure 3.1: Overview of World Generation System

second part then involves the positioning and drawing of these specific objects in the virtual scene. The specific objects for this project consist of parked cars and stop signs, which are drawn in the specific location they are detected in the input scene. In order to determine the position of these objects with respect to the input scene, the system will gather metadata about the specific objects in the input scene and transfer it to the virtual street scene application for drawing. These specific objects drawn together with the background scene constitute our complete virtual street scene. The structure and flow of this system is shown in Figure 3.1.

3.2 Virtual Street Scene Background

The first part of our virtual street scene application involves drawing the background of our virtual street scene as shown in Figure 3.2. In this portion of the system, the generic objects will be drawn to create the virtual scene. Generic objects are objects in the virtual scene that will always have pre-determined positions. This

is useful with world generation as there are certain aspects of a driving environment that do not change. In this project, these generic objects consist of roads, blocks, sidewalks, and buildings. We model our background on streets in San Francisco as many streets are laid out in a grid-like fashion. As a result, this world will follow a generic street scenario of having a two way street, at least one driving lane and one parking lane, a block with buildings on each side of the street, and a side walk on each side of the block. The units of measure for the virtual scene are in feet with a $1/10^{th}$ scale, i.e. a value of 1.2 units in the virtual scene translates to 12 feet in the real world.

As this world is meant to be a template of a scene rather than a set scene, there are multiple globally defined variables that a user can change in order to change different aspects of the static object. Each subsection will define which variables are associated with what changes to the world. Again, these variables are pre-defined as the background is meant to provide context rather than to be actual objects of focus. However, each of these variables can be defined in future works by the system rather than a user to make the background also specifically placed than generally placed.

3.2.1 Roads

Roads in the real world essentially consist of a large base and are defined by the lane markings and blocks surround them. As a result, the roads in this world consist of simply a large base which define the area of the world and the different lanes as shown in Figure 3.3a. To generalize the scene as much as possible, the world will always consist of a two way street. As a result, the road will always have solid yellow lines that mark the center of the road and solid white lines that mark the divide of the road from parallel parking. These lines are drawn as long, flat rectangles rotated respectively for each street.

Figure 3.2: View of fully generated world

One thing that does vary with roads are the number of lanes each road contains. In the real world, roads may contain as few as one lane to three or six lanes per side. As a result, in the code, a user can change the global variable `NUM_LANES` in order to control how many lanes each side of the street will contain. Lanes are defined by broken white lines drawn as repeated flat rectangles. Figures 3.3a and 3.3b show the differences between a single lane road and a multi-lane road. As default, the value is set to 1 to indicate a single lane road.

Another variable regarding lanes is `LANE_SIZE`. This variable controls how wide each lane is. As most roads have equally sized lanes, this variable controls the width of all lanes. The default value for this is 1 which correlates to 10 feet which we measured to be an average street width in San Luis Obispo.

(a) Single lane road.

(b) Multiple lane road.

Figure 3.3: Examples of roads with different number of lanes

3.2.2 Blocks

Blocks are simply separated flat areas of the world. They help denote the space for parallel parking and where buildings should be placed as well as are the base for sidewalks. For simplicity, the blocks are assumed to be square as we model them after grid-like blocks in San Francisco. As a result, the implementation of blocks are just large, slightly raised cubes. In addition, to generalize for this project, it is assumed that there are only four blocks in this world.

While assumed to be squares, it is not assumed that blocks are always the same size. As a result the variable `BLOCK_SIZE` defines how wide each block is. As this application draws objects from the center out, this variable denotes how wide half a block is. For example, the default value is 15 units wide which correlates to a total width of 30 units, or 300 feet in the real world.

3.2.3 Buildings and Sidewalks

Buildings are represented by apartment complexes. They stand on top of blocks and help define the sidewalks. While buildings almost always differ in shape and size, they are not the primary focus of this thesis so the buildings are generalized to

Figure 3.4: View with blocks

be simple apartments. The buildings are placed with respect to the blocks such that their entrances all face the street. To add more variety, the buildings vary with height depending on a random seed.

Sidewalks are defined by how much room is left on the block after the buildings have been placed. As sidewalk sizes vary, the variable `SIDEWALK_SIZE` defines how wide all of the sidewalks are. In order to make all the sidewalks uniform, this variable also affects the spacing between buildings to make sure they are also uniform. Figure 3.5 shows the addition of the buildings and sidewalks.

3.3 Positioning and Drawing of the Specific Objects in the Scene

The other part of this system consists of determining the position of the specific objects in the input scene and drawing them in our virtual street scene. The system takes in as input video taken from a front facing camera from a car on a road. This input video is then split into different frames which become the input scenes used to

Figure 3.5: View of fully generated world

create the virtual street scene. For each image, the system determines what objects are in the scene as well as where each object is located. This is done via segmentation and depth estimation in our metadata extraction step. This is then compiled in our metadata compilation step and transferred to the virtual street scene application for drawing.

3.3.1 Segmentation

The first step of this process is to determine what objects are in the image. This implementation uses the Mask-RCNN algorithm to do this [23]. As training a brand new network would take large amounts of data and time, He *et al.* provide users with a pretrained system. This pretrained system was trained off Microsoft’s Common Objects in Context (COCO) dataset, which is a general large-scale object detection, segmentation, and captioning dataset [28]. It contains images ranging from animals to people to vehicles. While general, the COCO dataset contains a large amount of data on cars and stop signs, allowing the pretrained system to perform accurate

segmentation on cars and stop signs. The COCO dataset contains around 23,304 images of vehicles for training, which proved to be enough images for training an effective model. From extensive tests, we found that the pre-trained model on COCO was able to have an average of 95% confidence that an object was a car when it detected a car in the image. As a result, rather than training a network from scratch, this project uses the pretrained system as it is accurate and suitable for the system's segmentation needs [23].

Mask-RCNN is simple in use as it takes in an image and outputs a visualization of the segmented image. In addition, it is easy to gather all the metadata needed for positioning each object in our virtual world. These pieces of data include the top left and bottom right corner bounding box coordinates, the label, the percent confidence, and the color of each of the segmented objects. One notable issue encountered is that the pretrained model was not trained specifically only on vehicles and stop signs, but on a variety of different objects. As a result, it would correctly segment the vehicles and stop signs in the scene, but would also segment other non-important objects with respect to this project, such as plants or birds. In order to gather metadata on just vehicles and stop signs, we use algorithm 3.1 to filter out unwanted objects and separate our desired object metadata into lists of vehicle and stop sign values. We found after extensive testing that an accuracy rating of 88% was high enough to eliminate outliers while low enough to ensure all the desired objects in the scene were segmented correctly.

3.3.2 Locating Position

After getting the metadata of the desired objects from the image, we can find the center position of the desired object using its pixel-space bounding box coordinates. This is done by adding the top left and bottom right corner bounding box coordinates

Algorithm 3.1 Mask-RCNN Filtering

```
1: for each image do
2:   labels, confidenceLevel, boundingBoxCoords, color = visualize(image)
3:   for each detected object do
4:     if label is a car or truck && confidence level > 0.88 then
5:       Add to list of cars
6:     end if
7:     if label is a stop sign && confidence level > 0.88 then
8:       Add to list of stop signs
9:     end if
10:  end for
11: end for
```

and dividing by two, as the origin of the image is located at the top left corner of the image. Figure 3.6 shows an example segmented image and one of the vehicle’s center.

One important aspect of Mask-RCNN is that it is able to create very tight bounding boxes due to it’s pixel based classification [23]. This allows the bounding boxes to stretch only as far as the furthest pixel classified rather than an estimate of the area that an object is in. Because of this, we can find the center of the object by calculating the center of the bounding box. This center value is then used in following sections as a reference point of where to estimate the depth of a object.

3.3.2.1 Position Intermediate Representation

The next step involves transferring this metadata to the metadata compilation step to assist in extracting the depth of each object in our scene. To do this, we created an intermediate text file called the Position Intermediate Representation (PIR) for

Figure 3.6: Segmented image and it's different given coordinates. Origin is located at the top left of the image with the x coordinates increasing moving to the right and the y coordinates increasing moving down. The depth is gathered by looking up the center value found in the segmented image in the depth map array.

our system to consolidate and pass on the metadata without much difficulty to parse or understand. The format of the file is shown in algorithm 3.2 where items in brackets are replaced by their actual values. After this file is generated, it is then sent to the metadata compilation step. This intermediate representation is simply our current best interpretation of transferring the metadata. In future implementations, alternative methods could be developed and used.

Algorithm 3.2 Position Intermediate Representation

- 1: {Width} {Height}
 - 2: {Number of cars}
 - 3: {Number of stop signs}
 - 4: {Label} {Bounding Box x1} {Bounding Box y1} {Bounding Box x2} {Bounding Box y2} {Color R value} {Color G value} {Color B value}
 - 5: ... (repeat for each object)
-

3.3.3 Determining Depth

We determined depth using a monocular depth estimation algorithm [18]. The algorithm takes in as input images and outputs a depth map for each. This depth map is then sent to the metadata compilation step where the depth of each object is extracted from the depth map by combining the two sets of metadata.

The first algorithm we used was monodepth [18]. However, one caveat of monodepth is it produces a depth map at a resolution of 512x256. This causes problems as PIR outputs pixel positions relative to the input image given to Mask-RCNN which are arbitrarily. In order to match these input images with the 512x256 constrained resolution output from mono-depth, a 2D coordinate transform is used to appropriately scale between the resolutions. However, this mapping frequently meant a loss of detail and proved challenging.

With monodepth not performing well, we looked for another monocular depth estimation algorithm and found monodepth2. Monodepth2 was released in 2019 as the successor to monodepth. It follows a similar structure to monodepth but made multiple significant improvements including the creation of larger resolution depth maps. This updated algorithm now allowed for depth maps of 1024x320 which more than doubles the amount of pixels and thus precision of the algorithm. Figure 3.7 shows the difference between the different depth maps.

While the same problems with monodepth applied to monodepth2 with regards to a constrained resolution, we found after considerable testing that resizing the input images to a resolution of 1024x320 provided enough detail for Mask-RCNN to still segment the image accurately while still giving a detailed depth map. This resolution is slightly smaller though similar to what Tesla’s cameras use for autonomous driving as they have a resolution of 1280x960 for their front and back cameras. This provides support that using a 1024x320 resolution is possible and feasible for detecting cars and depth.

Monodepth2 inherently produces a disparity map rather than a depth map. A disparity map is a map that shows the difference between two images and is used to gather depth as disparity is equal to inverse depth. As a result, we adjusted code in monodepth2 such that it performs this translation to depth and generates a depth map with depth values in it. This depth map is stored in a numpy array which will be sent along with the respective PIR to the metadata compilation step.

3.3.4 Metadata Compilation

As mentioned above, monodepth2 produces a depth map in the form of a numpy array. A numpy array is a specific file type made to be read and used in python

(a) Original Image

(b) Monodepth depth map

(c) Monodepth2 depth map

Figure 3.7: Comparison of Monodepth and Monodepth2

with the library `numpy`. In the metadata compilation step, we process the depth map and PIR to create another intermediate text file that consolidates and passes on the relevant metadata without much difficulty to parse or understand.

Compiling the metadata involves looking up in the depth map the depths for each of the objects enumerated in the PIR. As mentioned early, this will be done using the centers of each of the objects as the reference point and looking up in the depth map to find the depth value associated with the object. From multiple tests, we found that the most accurate calculation of the depths came from sampling the 5x5 grid of values around each center point in the depth map. Figure 3.6 shows what this process looks like.

While the depth map does contain depth values, the depth values are actually some scale of the actual depth of the object rather than actual depth. This results from `monodepth2` making an inferred guess of the depth from the environment rather than actual depth. `Monodepth2` only takes in as input an image meaning it does not know anything about the intrinsics of the camera that took the image. As actual depth depends on not only the environment but also the intrinsics of the camera, `monodepth2` will only be able to output a scaled depth prediction.

In order to find the scale of the depths, we conducted an analysis of the relationship between the actual depths measured versus the predicted depth. In order to do this, we first took pictures of some sample scenes from the world and measured the actual distances from the camera to each of the cars or stop signs. While the system will take in video as input, we decided to just use still images rather than video as we had trouble correlating objects in a video with actual distances as we did not have any special equipment that could determine distances in real time. In each scene we used, we instead used a tape measure to manually measure the distances from the camera to the cars or stop signs.

Figure 3.8: Actual depths vs predicted depths

After gathering the actual distances, we ran each scene through our metadata extraction step in order to get the depth values of each image. From there, we plotted the data of monodepth2 predicted depth values against the actual depth values. We then found the formula of the best fit curve, which represents the relationship between the monodepth2 predicted and actual depth. This formula is then given as input the predicted depth from monodepth2 and outputs the predicted depth our system uses for positioning the cars and stop signs in our virtual world. Figure 3.8 shows the relationship between the two sets. Formula 3.1 is the best fit formula with a R-squared value of 0.9493 (R-squared representing the ratio of explained variation to total variation), where x is the value from the depth numpy array. We then multiply x by 3.2808 to convert from meters to feet as the depth map outputs meters.

$$1.9356 * (x * 3.2808)^{1.4798} \quad (3.1)$$

3.3.4.1 Depth Intermediate Representation

After getting the depth, all relevant information is then written to a depth intermediate representation (DIR)¹, which is the intermediate text file that consolidates all the metadata so that it can be easily transferred to the virtual street scene application. The format of the file is shown in algorithm 3.3 where items in brackets are replaced by their actual values. An outline of compiling the metadata is shown in algorithm 3.4. Again, this intermediate representation is simply our current best interpretation of transferring the metadata. In future implementations, alternative methods could be developed and used.

Algorithm 3.3 Depth Intermediate Representation

- 1: {Width} {Height}
 - 2: {Number of cars}
 - 3: {Number of stop signs}
 - 4: {Label} {Bounding Box x1} {Bounding Box y1} {Bounding Box x2} {Bounding Box y2} {Color R value} {Color G value} {Color B value} {Depth value}
 - 5: ... (repeat for each object)
-

3.3.5 Cars and Stop Signs

In order to concisely represent cars and stop signs in the virtual world, a car class and a stop sign class are created. The car class consists of the center, the associated color from Mask-RCNN, the rotation, the depth, and the 3D position of the car. The stop sign class similarly consists of the center, the rotation, the depth, and the position of the stop sign. Here, the DIR is used to create arrays of the cars and stop sign classes.

The first step in determining where to position the objects in the virtual world is

Algorithm 3.4 Metadata Compilation

```
1: for each respective image do
2:   d = numpy depth array
3:   f = position intermediate representation
4:   for each object in f do
5:     c = center of object
6:     disp = d[c]
7:     depth = 2.1049 * (disp * 3.2808) ** 1.4998 {apply conversion formula 3.1}
8:     write to depth intermediate representation
9:   end for
10: end for
```

to determine if they are on the left or right side of the street. As it is assumed the input images are taken looking straight down a lane on the right side of the road, cars on the right side of the street should be located on the right side of the image and vice versa for cars on the left. With that fact, cars can be separated into left and right car arrays based on whether the center of the car is greater or less than the width of the image, which is given in the DIR. In addition, as only parked cars are considered, cars will always be in their respective parking lanes making positioning perpendicular to the axis going down the road easy to compute (x or z component of position). This whole process is done with stop signs as well as they will also always be located slightly past the parking lane on the sidewalk.

All cars and objects will always be at a constant and predefined height in the world allowing the y component of position to be easily calculated as well. The depth for the object is obtained from the depth intermediate representation giving the x or z component of position. With that, every object will have an x, y, and z coordinate for position and will then be drawn into the world accordingly.

(a) Low poly car system

(b) Low poly stop sign system

Figure 3.9: Example of systems used

Cars in the 3D world will be represented by the same car model shown in Figure 3.9a. This low poly model consists of 993 triangles and only takes up about 135 KB of memory making it a suitable system that takes up little space. Stop signs will also be represented by the same stop sign model shown in Figure 3.9b. This model consists of 100 triangles and takes up 82 KB of memory.

The finished virtual scene combines the static and dynamic objects of the scene together. Figure 3.10 shows an example of a finalized generated world.

Figure 3.10: Example Final View

Chapter 4

RESULTS AND VALIDATION

Analysis of this system will look at the accuracy of our system, the memory intensity of our system compared to point clouds, and the time consumption of our system. The primary ways to analyze the accuracy of our system are through the visual and quantitative accuracy of the placements of the models in the 3D world. As it is made to represent the real world, it is important to see how close or far off the positions of models in the world are to the actual cars in the real world images. To analyze the memory intensity of our system, we will look at the size of files transferred and the time it takes to do so.

4.1 Data Gathering

The first choice for data gathering was using video as that is the input that the model takes. However, after much trial and error, we found that we could not correlate frames in the video with distances in the real world as we did not have any special equipment that could determine distances in real time. As knowing the distance in each frame is important for getting results and validation, we instead opted to manually use a long tape measure and capture images at set distance intervals instead of use video. Images were taken every two feet from the first frame totaling to 95 images. At the first frame, we manually measured the distance to each car from the camera to get our baseline distances. With this baseline, we were able to calculate the distances to each car from the second frame as the baseline and the second frame was two feet apart meaning that the distances of the cars from the second frame was

two feet less than the baseline. This logic can be applied to all the frames taken giving us the distances to each car from each frame.

The accuracy of this model will be determined by comparing virtual scenes with their real image counterparts. The tested images are images that have not been used to create the model in order to get an unbiased result. These images were taken by an iPhone XR and cropped and resized to a size of 1024x320 in order to be used with the monodepth algorithm as explained in Section 3.3.3.

As the scope of the dynamic objects in this project are just parked cars and stop signs, the relevant information is defined as parallel parked cars and stop signs closer than 200 feet. To focus on that, these images were modified to remove extraneous information such as cars parked in driveways or cars further than 200 feet away from the first frame by coloring them black. These images were cropped and resized manually using the open source software GIMP to contain the parked cars with a center focus on the horizon as shown in Figure 4.1.

4.2 Analysis

4.2.1 Visual Accuracy

To test visual accuracy, each of the test images were run through the world generation model and outputted for comparison between the original image and the generated world. Figures 4.2, 4.3, 4.4, and 4.5 show the results for different test images. One thing to note is that the model is just dynamically placing the cars, but not the buildings or roads and as such the scene of the buildings and roads do not change to fit the real world scene.

As can be seen, the generated world representations closely resemble the car po-

Figure 4.1: Example original test image (top) and modified test image with extraneous information blacked out (bottom).

(a) Original

(b) Segmented

(c) Generated virtual scene

Figure 4.2: Good scene representation as all objects match their respective positions in the original.

(a) Original

(b) Segmented

(c) Generated virtual scene

Figure 4.3: Good scene representation as all objects match their respective positions in the original.

(a) Original

(b) Segmented

(c) Generated virtual scene

Figure 4.4: Incorrect scene representation as there is supposed to be a car behind the pink car on the right.

Figure 4.5: Incorrect scene representation as the cars are all too close to the camera.

sitions of the input scenes. There are good results such as Figure 4.2 and 4.3 with object positioning that strongly match the object positions in the original image.

However, cars that are heavily occluded or blocked from view tend to not be drawn as seen in Figure 4.4 as there should be three cars on the right hand side but only two were drawn. This is not too concerning as these cars are normally occluded by another car that we detect in the image, which is more pertinent than the occluded car. Still, our system sometimes fails with certain images such as with Figure 4.5 as the cars are too close to the camera. This can be due to differences in lighting such as being shrouded by shadows or by being too reflective in the sun.

Overall, our system is able to produce a good representation of what the real world looks like visually, though it does struggle in certain conditions.

Figure 4.6: Graph showing the predicted distances against the actual distances.

4.2.2 Quantitative Accuracy

For quantitative accuracy analysis, each of the test images was run through the model up to the depth intermediate representation in order to gather all the predicted distances.

Figure 4.6 shows how the data was spread out with respect to the predicted and actual distances. There were many data points that had a little variation before the 50 feet mark, but the variation increased as the distances got further. Figure 4.7 shows another representation of this data through a five-number summary of the predicted distance difference grouped into ten feet intervals. These values are calculated by taking the absolute value of the difference between the predicted distance and the

Figure 4.7: Graph showing a five-number summary of the predicted distance difference error. The lines for each column shows in ascending order the minimum, first quartile, median, third quartile, and maximum values for the data in the 10 foot range.

Figure 4.8: Graph showing the ratio of samples to the amount of difference for each increment of 10 feet. The orange bars represent how many samples were in that 10 feet range while the blue line shows the average difference between the actual and predicted distances for cars in each respective distance interval. The yellow line shows the total average distance over all the prior intervals.

actual distance. In ascending order, the horizontal lines for each column shows the minimum, first quartile, median, third quartile, and maximum values for the data in each interval. This graph shows that as the distance interval gets higher, the amount of variation in the data increases and the distance difference gets worse overall. Table 4.1 further emphasizes this point as the mean and standard deviation of the intervals increase as the distance interval gets larger.

Figure 4.8 shows the ratio of samples to predicted distances. It also shows that the average difference increases rapidly as the cars get further away from the camera. This is to be expected as cars further away provide less details in images, which in turn give the algorithm less information to estimate the distance from the camera.

Algorithm 4.1: Table showing the distance intervals and their respective standard deviations and means

Distance Intervals	Mean	Standard Deviation
20	0.887787	1.070232
30	1.644558	1.806408
40	2.028592	2.083097
50	3.234387	2.322738
60	3.921423	4.499874
70	6.746362	4.033291
80	6.00456	4.337117
90	7.047249	5.40102
100	7.522863	6.343805
110	9.973285	12.00415
120	15.03949	12.61475
130	13.65772	13.20123
140	19.62751	11.34206
150	20.44052	7.106236
160	11.0536	7.749055
170	18.81354	10.74121
180	6.836983	3.4114
190	10.19474	5.174792
200	7.218426	2.957417

Overall, the model has a total average distance difference of ± 7.466 feet as shown in Figure 4.8. While this is not a great accuracy as the length of an average car is about 10 feet [36], cars closer to the camera are more important compared to cars far away. Closer cars are seen more compared to cars far away as it is harder to detect cars further away than close by. This is reflected in Figure 4.7 as the majority of the data at distances less than 80 feet have a distance difference of less than five feet. In addition, Figure 4.8 shows that there is a much higher density of samples at distances less than 70 feet than there are at distances greater than 70 feet. Cars further away also have less impact than cars closer to the camera (or the car in a real life application) as the immediate concern is not to crash into cars.

Cars further away will also eventually get closer to the camera as the camera moves forward. As this happens the car depth will continually be updated and readjusted over time. As a result, we can focus on the accuracies of the model from smaller distances than 200 feet away. Table 4.2 shows that the range from 10-70 feet provides a better ratio of error to distance as ± 3.39 feet spans less than half the length of a typical car and as 70 feet provides the possibility for seven cars to be on parked on each side of the street. This is also supported Figure 4.8 as there are a high number of samples while still having a low error rate at distances less than 70 feet away. Overall, the model has an average distance difference of ± 3.39 feet and an average error rate of 0.0704%. However, ± 3.39 feet is still a considerable amount of error with respect to cars making this system not perform well enough to be used in a real world application.

Algorithm 4.2: Distance differences based on intervals

Distance Intervals	Average Distance Difference	Average Total Distance Difference
	Per Interval	up to Interval
20	0.88779	0.88779
30	1.64456	1.53645
40	2.02859	1.82594
50	3.23439	2.31348
60	3.92142	2.70964
70	6.74636	3.45011
80	6.00456	3.84605
90	7.04725	4.26359
100	7.52286	4.59461
110	9.97329	5.03802
120	15.03949	5.67569
130	13.65772	6.08242
140	19.62751	6.62091
150	20.44052	7.07066
160	11.05360	7.19620
170	18.81354	7.51980
180	6.83698	7.50858
190	10.19474	7.55202
200	7.21843	7.54671

4.3 Memory Intensity

The system created also must consider the memory intensity of the representations. As a car might not have a driver, there may be situations when the autonomous vehicle may need to communicate with a remote operator or personnel to aid in navigation or debugging of problems. This leads to the importance of transferring data efficiently and wirelessly as the vehicle will be moving around and most likely not connected to any physical connection. As a result, we will be looking at mobile data upload rates and the size of the representations when transferring data.

Currently, most mobile data networks operate at the 4G LTE network. However, there are companies that are pushing and in the process of adopting to a 5G network. While 5G has not been adopted yet and the mobile upload rates are still theoretical, the speeds between these are vastly different and so we will look at the performance over both these networks. Currently, Verizon has the fastest mobile data upload rate with a maximum speed of 65.2 Mbps and an average speed of 19.0 Mbps [40]. On the other hand, a 5G network can theoretically have upload speeds of anywhere from 300 Mbps to 1 Gbps, making it at least 5 times faster than 4G LTE [13].

Figure 4.9 is a mesh of a point cloud of the environment where the test images were taken created by a software called Meshroom. While not a perfect point cloud of the environment, it is a good representation of the environment that we can use for analysis. This point cloud has a memory size of 29.2 MB. If transferring the data to a remote operator or personnel using this representation, the entire point cloud would need to be transferred.

Comparatively, our system would only require the transferring of the DIR as the remote operator or personnel can have the virtual street scene application which only

Figure 4.9: The textured and untextured meshes created using *Meshroom*.

requires the DIR to place object in the scene. A single DIR file has a memory size of 412 bytes. To more accurately compare the size to the point cloud as our system would need multiple DIR files to cover the distance that a point cloud covers, the folder of DIR files our system needs to represent the entire environment has a memory size of 24.9 KB.

If transferring using Verizon’s 4G LTE average mobile data upload rate of 19 Mbps, the point cloud of 29.2 MB would take around 12 seconds to fully upload the data. On the other hand, the DIR folder of 24.9 KB would take much less than a second to upload. However, if we use the theoretical 5G network speeds, both representations would be upload almost instantly even if the speed is at the low range of 300 Mbps. This brings another point that when 5G networks are widely adopted, point clouds could be easily transferred and the memory intensity of point clouds would almost be nonexistent. Still, as we are not there yet, the transferring of our representation is much more memory efficient and thus time efficient than the transferring of a representation using point clouds.

4.4 Time consumption

The time it takes for our system to make a full pass is also very important. The main areas that we will look at for this is the virtual street scene application, monodepth2, and Mask R-CNN. The compilation of the meta files are done almost instantaneously as they are just doing simple math and lookups so we can ignore the amount of time they take.

Our virtual street scene application is run in C++ and is currently run through Visual Studio. After much experimentation and testing, we found that the application takes 5.5 seconds to begin running. However, after it starts running, it is able to switch between different DIR files almost instantaneously to create new scenes. As a result, we can conclude that the virtual street scene application contributes a negligible amount of time to the system as the application can be started ahead of time and just read in a DIR file when it comes.

Monodepth2 is the algorithm we used for gather depth from an image [19]. The time consumption it takes is split into two different parts. The first part is loading a local pre-trained model that it uses to perform the depth estimation while the second part is the actual depth prediction. This loading process takes on average 2.01 seconds and must be done every time the depth function is called. However, much like with our virtual street scene application, the loading process can be done ahead of time as it is not necessary to do for input frame from our input video. The depth prediction portion then is what contributes to the time the system takes. After much testing, we found that it took an average of 0.88 seconds when given one image, but took less time when given more than one image with an average of 0.178 seconds when given 60 images to process. This may be due to the time it takes to cleanup after processing all the images. As we can theoretically stream images to the algorithm from the video

stream, we can alter the code to make it take in multiple images at a time such that it does not need to perform any cleanup. Thus we can say that monodepth2 takes around 0.168 at best if constantly processing the video stream while taking around 0.88 seconds if processing one image at a time.

While our virtual street scene application and monodepth2 seem fairly adept at processing the data in a short amount of time, Mask R-CNN, the segmentation algorithm we used, becomes a bottleneck of our system’s performance as it is very slow at processing and segmenting images. He et al. state that the system can process an image at 200ms on a single GPU [23]. This means that the model is rated to work at 5 frames per second if given a single GPU. While possible to add more GPUs and perform more optimizations, the best result users have found were to get 15-18 frames per second [23].

As a result, our system will be limited to operating at best 15-18 frames per second due to the bottleneck created by Mask-RCNN. As a result, our system would not be able to be efficiently used in real-time to create a virtual scene representation. However, since Mask R-CNN is the only bottleneck to our system since the virtual street scene application and monodepth2 add little time to the system, using a faster more efficient segmentation algorithm may make this system be able to run in real-time.

Chapter 5

CONCLUSION

While the model created does work, the error of a couple feet still leaves much more precision to be desired. In addition, the world generation itself is still very constrained as it is limited to just parked cars and stop signs. This section will detail the different mistakes and challenges encountered in this thesis as well as possible steps to improve the model.

5.1 Future Work

5.1.1 Depth

When first determining how to gather depth from the scene, we encountered a couple promising methods for depth acquisition. These included the use of monocular image algorithms, stereo images, photogrammetry software, or LiDAR. Each have their own merits and downfalls as will be described in the following sections.

5.1.1.1 Monocular Depth

We picked monocular image depth estimation as our depth estimation method as that seemed to be the most simple and generally applicable method of gathering depth at the time. It was very appealing that any user could use their phone or any other camera device to get a depth map without any other specific equipment. This ease would make it so that the model we created could be used with any vehicle without any other proper setup besides the ability to take clear pictures of the road.

In addition, while monocular depth estimation is an ill-posed problem as explained in Section 2.3, researchers are constantly exploring and creating better algorithms every year, sharing their findings in ongoing competitions held by creators of prominent datasets like the KITTI dataset [14, 42]. These algorithms have also been improved to be able to run and create depth maps in less than a second, allowing them to be used in real time.

For this thesis, we used the publicly available algorithm monodepth2 to gather monocular depth maps. While easy to use and producing some good results, monodepth2 is not currently the best algorithm for monocular depth estimation. According to the KITTI dataset competition, the current best methods for monocular depth estimation are algorithms called BTS(Big to Small) and DORN(Deep ordinal regression network). These algorithms were able to have a absolute relative error of 9.04% and 8.78% respectively when tested on the KITTI dataset [14, 42]. These values may decrease even more if optimized for the types of scenes used in this thesis as monodepth2 normally has an absolute relative error of 11.22% but when fitted to work with our data, achieved an absolute relative error rate of around 8% overall and 7% when constrained to less than 70 feet. However, these algorithms are not available for public use as BTS was submitted anonymously and has no paper or documentation associated with it and as DORN only publicly provides a skeleton structure for research purposes rather than the complete built framework.

Despite not being the best algorithm, our results still show that the accuracy of monocular depth estimation is not precise enough for actual use. While implementable, an error of ± 3 feet is still too large to be confidently used in a real life situation. Any system dealing with vehicles must be very accurate in order to avoid collisions or other miscalculations. In addition, even the better algorithms still contain error that is not significantly lower than monodepth2. This leads to the conclusion

that monocular depth is not the best method of depth estimation at this current time. However, this may change in the future as algorithms from year to year have on average a lower error rate and that trend seems to be continuing [14, 42]. As it was publicly available and as we had resource constraints, monodepth2 was determined to be the best use for this project at this time, but in the future additional measures should be used, some of which are explored in the following sections.

5.1.1.2 Computer Stereo Vision

Computer stereo vision is the process of extracting information from images taken from a pair of cameras displaced horizontally from each other. Using left and right images, algorithms compute depth maps by tracking features in both images and using known data such as the focal length and the distance between the two cameras that took the left and right images. As a result, stereo depth maps are able to be so accurate that the error metric commonly used is the percentage of pixel outliers in the depth map rather than the difference between the predicted and actual depth, as most algorithms give close to exact depth. KITTI has another ongoing competition for stereo depth and the best algorithms submitted were able to have an error of 1.76% of pixels as outliers [30, 29]. However, these algorithms are, much like the monocular depth algorithms, not readily available for public use.

One public way to get stereo depth maps is through OpenCV [6]. OpenCV is a public python library that has support for many different types of image processing including computer stereo vision. In order to use it properly, OpenCV must first take in camera parameters for calibration. This is necessary as cameras inherently warp photos to different extents depending on the focal length of the camera. Having the camera parameters allows OpenCV to unwarp the photos so that it can make a more precise depth map. This is normally done by taking pictures of a checkerboard from

multiple different angles and using OpenCV's `cameraCalibration` function. With that information, OpenCV can then read in the stereo photos, undistort them, and run them through its `StereoSGBM` object. The `StereoSGBM` performs feature matching by using block matching, or going through both image in blocks and finding matching blocks. This function itself takes in a number of parameters to alter and adjust the result of the disparity map to make it smoother and more accurate. A quick lookup of the documentation for this object provides more information to all the parameters, but some notable parameters are `minDisparity`, `numDisparity`, and `SADWindowSize`. `minDisparity` equals what the smallest disparity value is for the foreground while `numDisparity` equals the largest disparity minus the smallest disparity to determine the max disparity value for the background. `SADWindowSize` determines how many pixels to match when feature matching between the two images.

Another open source algorithm called DMAG (Depth Map Automatic Generator) also works quite well with making depth maps from stereo images [9, 33]. It performs differently than OpenCV as it does not perform feature matching through block matching but rather through cost volumes. Cost volumes are computed by comparing all pixels in an image to the pixels of other other image and storing the difference, or cost. While simple, it is effective and produces accurate depth maps. However, it does take some time if the images get bigger as it is $O(n^2)$. Still, it proved to be easier and more accurate for us to use than OpenCV as it figured out many of the values for our images automatically. DMAG is able to run off both a command line interface as well as through a very nice GUI called *StereoPhoto Maker*. Figure 5.1 shows a resulting depth map compared to the `monodepth2` depth map of the same image.

One issue with creating stereo depth maps is that the images must be stereo meaning that two cameras should be in place for this system to work. Computer

(a) Left Image

(b) Right Image

(c) StereoPhoto Maker Depth Map

(d) monodepth2 Depth Map

Figure 5.1: Two input images left and right images and the corresponding depth maps created. Both depth maps are of the left image, but StereoPhoto Maker uses both left and right images to compute it while monodepth2 only uses the left image.

stereo vision relies heavily on the fact that the two images are perfectly parallel to each other. When we tried stereo depth estimation towards the beginning of our research, we did not make sure that the images were perfectly parallel to each other giving us poor results as we only had one camera. A simple setup for producing precise stereo depth maps is to have two cameras attached to a wooden board or some other flat solid object and have a remote take a picture from both at the same time. This setup can then be mounted onto a real or model car and used in place of the monocular depth map in this system.

5.1.1.3 Photogrammetry Software

Another possible way to gather depth from images is to use photogrammetry software, which gathers measurements from photos. In many cases, these software create point clouds from a series of images that describe what a scene looks like. One software that we were able to use and work with is called *Meshroom*. *Meshroom* takes in images from various angles about a scene and creates a mesh from it as shown in Figure 5.2. This mesh was made up from the 95 image used for testing. Each of the white squares represent where it thought an input photo was taken.

The mesh created has a solid resemblance to the actual scene and while there are spots missing from the mesh, the overall shapes of the cars are present and the distances of each car appears similar to what the distances were in real life. By combining the mesh with some image segmentation, this can provide a very detailed depth map of the scene.

The main caveat to this method is the amount of time it takes to process the data. Figure 5.2 took around 30 minutes to complete processing all 95 which is quite long compared to *monodepth2* which took around three to five seconds to create a depth

Figure 5.2: The textured and untextured meshes created using *Meshroom*.

map of all those images. As a result, this may not be as good of a method if time is under consideration.

5.1.1.4 LiDAR

LiDAR is another very powerful tool for depth acquisition. It is one of the most accurate pieces of equipment for gathering distances as the lasers provide pinpoint accuracy and as there is little variability in the data gathered as light always travels at the same speed. The big downside to LiDAR systems then are their price as they can get very expensive. However, there are two different types of LiDAR, 2D LiDARs and 3D LiDARs. 2D LiDARs are much cheaper than 3D LiDARs and are much smaller as well so if there is a budget, getting a 2D LiDAR could greatly aid in depth acquisition. However, it is important to note that different LiDAR models do vary quite a bit in the range they can scan. As this model is intended to be used in a real world environment, a LiDAR used with this model should have a range of at least 100 feet if not more to create a decent scene reconstruction.

If a LiDAR of that range is not available, another alternative might be to get a LiDAR with a smaller range and combine that with any of the previously mentioned methods for depth estimation. In doing this, the further distances may have more error, but as the camera approaches the further away objects, they will have their distances recomputed and rectified once they get in range of the LiDAR.

We experimented with using a 2D LiDAR called RPLIDAR A2 for depth acquisition, but realized that the LiDAR had a range of at max 60 feet and only reliably acquired depths 10-15 feet away. In addition, it struggled to gather depths of reflective objects, which at most times were the cars we were trying to get depths of. As a result, the LiDAR proved to not work very well with this system. Still, LiDAR is a very powerful piece of equipment and future work could include integrating LiDAR into the system. This could be in the form of a scale environment where the objects being measured are not further than what the LiDAR can measure on average.

5.1.2 World Generation

Currently the model only supports parallel parked cars and stop signs. This is quite constraining and only applies to certain real life situations. As a result, this model will not be able to be used in many different scenarios. Some future steps would be to expand the world generation to also dynamically generate perpendicularly parked cars, cars in driving lanes, stop lights, pedestrians, block lengths, and buildings. However, the most plausible and pertinent next step would be to dynamically generate perpendicularly parked cars and cars in driving lanes.

With world generation also comes the future step of implementing this system on a real life vehicle. This could be done with actual vehicles on the road or with scale model cars. While both are feasible in the future, implementing this system on

scale model cars will be a better option as it allows the system to be in a controlled environment. That way the environment can be changed to try out different scene scenarios that may be difficult to make happen when in an actual vehicle on the road.

5.1.2.1 Data Gathering Challenges and Tips

Continuation of building up this system will inevitably require more data gathering whether that be in the form of expanding the database or in the new forms of data. Data gathering was not an easy task to do for this project as it took much time and trial and error to get reliable, usable data. The first criteria for the data is to have images and depths associated with each object in the image. Our first thought was to use video as that is the way the model will be given data and as that provides a large amount of image data that can be used. However, the other criteria is to get the distances associated with each object in the video. As we did not have any equipment that could detect distances in real time, we would have to somehow associate each image in the video with manually measured or calculated distances. This proved to be quite difficult causing us to opt for another option which was to manually use a long tape measure to measure the distances for each car. Then we would capture images at set distance intervals instead of using video.

To do this, we started at where we would take our first image and measured the distance to every car in the scene as we were dealing mostly with parked cars. After measuring that distance, we extended the tape measure to the furthest distance it could go and began to capture images every two feet. This way, we would be able to associate the distance in every image we took as the measured distance to an object minus the image sequence number times two feet.

One difficulty we realized after our first few data collections was that it is impor-

tant that all the images be at the same height and centered around the same point. By having all the images be at the same height and centered, the only changing factor is the z distance rather than having the model also try to accommodate for x and y changes. As the end goal is for this to be used in a car where the x and y coordinates of the images should not be changing too much, it is important to try to keep those factors in the same way. However, since we had to manually measure distances and take pictures, it was not very feasible to take pictures from in our car and make sure the car moved in two feet intervals. As a result, we had been taking pictures using our phone without a tripod and while we did try to keep the phone leveled at the same height and focused at the same point, we was not able to perfectly do so. This caused images to sometimes be at slightly different angles or different height which in turn affected the depth algorithm’s accuracy. We would recommend then if in the future images must be taken manually to use a tripod and a camera. With respect to cameras, it is also important to have images that have FOV’s similar to human vision so that the model will generate a world that is similar to what a human would see.

5.2 Conclusion

In this thesis, we proposed a model that would take in video as an input and generate a virtual scene that would be a representation of a real world scene. The virtual scene would be a less memory intensive and more visually understandable representation of the scene than a sparse point cloud that is commonly in use today. This is accomplished by replacing objects from the scene with low poly geometric proxies in the virtual scene. In order to focus the scope of this endeavor, we focused on only using parked cars and stop signs as the relevant dynamic objects int the virtual world.

The model is split into gathering the metadata of the relevant objects in the video and the actual virtual street scene application. The metadata gathering is done by inputting still frames from the video into an image segmentation algorithm and a depth estimation algorithm. The outputs of these are then combined to find the position of each object in the image. This metadata is then sent to the virtual street scene application which draws out the the dynamic and static objects. It is here that the static portion of the model happens as well with the road, blocks, sidewalks, and buildings statically generated in the scene.

While the generated world does place cars in positions that resemble the scenes they were given, the model proved to not work very well as the predicted depth compared to the actual depth of the objects in the scene proved to cause more error than desired. The depth algorithm used was monodepth2, which is a monocular depth estimation algorithm. It gave up an average error 8% with a slightly lower error 7% when dealing with distances less than 70 feet away. This correlated to an average error of ± 7.46 feet and a lower error of ± 3.39 feet when dealing with distances less than 70 feet away. While these results are not terrible, the results are not precise enough to be confidently used in a real life scenario.

Still, there are some very feasible and plausible future steps that could be taken to build off this model. As depth proved to be a challenge, we laid out some other alternatives to monocular depth estimation such as stereo depth estimation, photogrammetry, and LiDAR. Each one has their own benefits and drawbacks, but it seems that stereo depth estimation can provide the best result with the least amount of extra work needed to adapt it to the current model.

5.2.1 Vision Moving Forward

This project provided a proof of concept for a virtual street view visualization system. It has limitations, but shows that there is the possibility for a system that can take as input video and create a virtual world that is not too memory intensive but also visually coherent.

Future iterations of this project should start on a smaller scale than what this project ended up at. One possible direction could be a move to using model cars rather than actual cars. Model cars make it much simpler to create controlled scenarios for data gathering and testing. In addition, they cost much less to acquire and can sustain more mistakes than a real-life car can.

Model cars are also simpler to update than real-life cars as they are compact and portable. This allows for ease of trying out different depth algorithms and modifications. We used monocular depth estimation as it was simple and required only a single forward facing video as input. However, it did not work out very well and was a bottleneck for the success of the system. As monocular depth estimation did not work well enough, stereo depth mapping is another easy but better algorithm for estimating depth.

It was very difficult to perform stereo mapping with one camera, as the photos needed to be parallel and taken at relatively the same time. Even if we had two separate cameras, it would still have been difficult to have consecutive stereo images at the same height and angle. However, stereo depth mapping is relatively straightforward with model cars as the two cameras can be mounted directly on the car, making them parallel and always taking images at the same angle and direction. This would allow us to create precise and detailed depth maps without too much effort as the stereo depth algorithms are not too difficult to implement as shown in Section 5.1.1.2.

Other possible next steps to improve the proposed system would be to support more specific objects to place in our virtual scene. Currently, the system only supports parallel parked cars and stop signs. However, most scenes usually have other cars driving on the road as well. This scenario is not too different as a scenario with a parallel parked car as the only difference in each frame is that the car is in a lane different than the parking lane. As a result, a way to include moving cars in the system would be to let the system segment the scene as how it is doing right now, but to also perform segmentation on the roads to see how many lanes are present in the scene. This type of segmentation is called lane segmentation and has had very high success rates as well as many open-source pre-trained models such as SCNN-Tensorflow [31]. With the segmented moving cars and roads, the system can then match which lane each car is in using the center point of each car and computing which segmented lane that center is in. That information can be passed to the virtual street scene application where the moving cars will be drawn with respect to the position they are at in the input scene. This can furthermore be extended to supporting adding pedestrians, stop lights, and more into the scene.

Finally, in terms of the visualization of the scene, the current system outputs geometric proxies of cars and stop signs in an arbitrary color. A future step would be color the cars and objects in the scene the same color that they are in the real world. This can be done by first segmenting the input image and finding the color of the object by sampling the color of either the whole object as Mask R-CNN performs pixel classification, or the pixels around the center of the object. That color will be much closer to the actual color of the object than any arbitrary color will be. This extracted color can then be sent through the system to the virtual street scene application, where it will then draw the object in its correct color. This way, the scene will look more like the actual scene rather than having objects with random colors.

BIBLIOGRAPHY

- [1] 'phantom auto' will tour city. *The Milwaukee Sentinel*, Dec 1926.
- [2] Sae international releases updated visual chart for its "levels of driving automation" standard for self-driving vehicles, Dec 2018.
- [3] About us, 2019.
- [4] J. R. Anderson. The adaptive nature of human categorization. *Psychological review*, 98(3):409, 1991.
- [5] A. Bhoi. Monocular depth estimation: A survey. *CoRR*, abs/1901.09402, 2019.
- [6] G. Bradski. The OpenCV Library. *Dr. Dobbs's Journal of Software Tools*, 2000.
- [7] M. Burns. 'anyone relying on lidar is doomed,' elon musk says, Apr 2019.
- [8] O. Cameron. An introduction to lidar: The key self-driving car sensor, May 2017.
- [9] U. Capeto. A quick guide on using stereophoto maker (spm) to generate depth maps, May 2019.
- [10] Comma.AI. How does openpilot work?, Jun 2017.
- [11] L. Dormehl and S. Edelstein. 10 Major Milestones in the History of Self-Driving Cars, Oct 2018.
- [12] P. Falkingham. Acquisition of high resolution three-dimensional models using free, open-source, photogrammetric software. *Palaeontologia Electronica*, 15:1T:15p, 01 2012.

- [13] T. Fisher. 5g speed: How to understand the numbers, Jun 2019.
- [14] H. Fu, M. Gong, C. Wang, K. Batmanghelich, and D. Tao. Deep ordinal regression network for monocular depth estimation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2002–2011, 2018.
- [15] R. B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.
- [16] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.
- [17] GISGeography. A Complete Guide to LiDAR: Light Detection and Ranging, Feb 2018.
- [18] C. Godard, O. Mac Aodha, and G. J. Brostow. Unsupervised monocular depth estimation with left-right consistency. In *CVPR*, volume 2, page 7, 2017.
- [19] C. Godard, O. Mac Aodha, and G. J. Brostow. Digging into self-supervised monocular depth estimation. *CoRR*, abs/1806.01260, 2018.
- [20] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
<http://www.deeplearningbook.org>.
- [21] E. Hall. Hands-on with comma.ai’s add-on level 2 autonomous tech, Aug 2018.
- [22] A. J. Hawkins. Gm will make an autonomous car without steering wheel or pedals by 2019, Jan 2018.
- [23] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

- [24] J. Hughes. Alphabet’s waymo is already running level 4 self-driving cars in arizona, Nov 2017.
- [25] Jianbo Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, Aug 2000.
- [26] K. Korosec. Tesla autonomy investor day is here; here’s how to watch, Apr 2019.
- [27] A. Kundu, Y. Li, and J. M. Rehg. 3d-rcnn: Instance-level 3d object reconstruction via render-and-compare. In *CVPR*, 2018.
- [28] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.
- [29] M. Menze, C. Heipke, and A. Geiger. Joint 3d estimation of vehicles and scene flow. In *ISPRS Workshop on Image Sequence Analysis (ISA)*, 2015.
- [30] M. Menze, C. Heipke, and A. Geiger. Object scene flow. *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)*, 2018.
- [31] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang. Spatial as deep: Spatial cnn for traffic scene understanding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
- [32] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.
- [33] C. Rhemann and C. Rother. Fast cost-volume filtering for visual correspondence and beyond. In *CVPR*, January 2012.

- [34] F. Rohrbach. Point density and point spacing, Oct 2015.
- [35] L. Rosasco, E. D. Vito, A. Caponnetto, M. Piana, and A. Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.
- [36] J. S. Average car length - list of car lengths, Jan 2019.
- [37] S. Saha. A comprehensive guide to convolutional neural networks-the eli5 way, Dec 2018.
- [38] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1-3):7–42, 2002.
- [39] J. Schauer. Robotic 3D Scan Repository, Sep 2017.
- [40] S. Segan. Fastest mobile networks 2018, Jun 2018.
- [41] D. Soni. Supervised vs. unsupervised learning, Mar 2018.
- [42] J. Uhrig, N. Schneider, L. Schneider, U. Franke, T. Brox, and A. Geiger. Sparsity invariant cnns. In *International Conference on 3D Vision (3DV)*, 2017.
- [43] A. Vance. \$800 million says a self-driving car looks like this, Jul 2018.
- [44] Y. Wang, Z. Lai, G. Huang, B. H. Wang, L. van der Maaten, M. Campbell, and K. Q. Weinberger. Anytime stereo image depth estimation on mobile devices. *CoRR*, abs/1810.11408, 2018.
- [45] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4):611–629, Aug 2018.

- [46] A. Zanfir, E. Marinoiu, and C. Sminchisescu. Monocular 3d pose and shape estimation of multiple people in natural scenes - the importance of multiple scene constraints. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.
- [47] D. Zermas, I. Izzat, and N. Papanikolopoulos. Fast segmentation of 3d point clouds: A paradigm on lidar data for autonomous vehicle applications. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5067–5073, May 2017.
- [48] G. P. Zhang. Neural networks for classification: a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 30(4):451–462, 2000.
- [49] J. Zhang, M. S. Ramanagopal, R. Vasudevan, and M. Johnson-Roberson. Listereo: Generate dense depth maps from LIDAR and stereo imagery. *CoRR*, abs/1905.02744, 2019.