computer graphics & vision

Inffeldgasse 16/2, A-8010 Graz

www.icg.tugraz.at

TU Graz

LiDAR resolution simulation in synthetic training data for 3D object detection

Phillip Stranger

Institute of Computer Graphics and Vision
Graz University of Technology, Austria

Bachelor Thesis
Supervisor: Prof. Dr. Horst Bischof
Graz, May 30, 2021

Contact: Phillip Stranger, phillip.stranger@student.tugraz.at

Abstract

3D object detectors need large amounts of annotated training samples to reach high accuracies, but generating these annotated training samples is expensive and labourious. This is the incentive to try to generate these datasets synthetically. Synthetic data describes data taken from virual environments like game engines. In this work we generate point clouds from the depth images of the Apollo Synthetic dataset to build a dataset on which a 3D object detector is trained. We show the impact the correct simulation of the LiDAR has on the accuracy of the 3D object detector. Additionally we simulate real world effects like noise and drop-out to further increase the accuracy of the 3D object detector.

Keywords: *3D object detection, synthetic data, LiDAR simulation*

Contents

1 Introduction

As car accidents are still, well over 100 years after the invention of the car itself, a common occurrence, the need for self-driving cars is apparent. According to the WHO (World Health Organisation)[1] yearly road fatalities are approximated to 1.35 million. With self-driving cars this number could be reduced greatly and therefore the car could become a safe and effortless way of transport. This would also enable the usage of the time currently taken up by driving the car in a more productive way. So with the creation of a system capable of securely driving a car the process of travelling by car would become safer and the time spent in the car could be used in a more meaningful way.

The autonomy of cars is classified into 6 levels by the Society of Automotive Engineers (SAE)[2]. The levels range from fully manual (Level 0) to fully automated (Level 5). Levels 0 to 2 characterize the Levels of automation where the driver monitors the environment. Starting with Level 3 all the way to Level 5 describe the Levels where the automated system monitors the environment.

Current comercially available autonomous driving systems, such as the Tesla Autopilot[3], fall into Level 2 and to reach higher levels of autonomy different techniques to monitor the environment need to be implemented. An important task is the detection of objects present in the environment, which currently is often done on 2D image data captured by multiple cameras. This works quite well when detecting objects, but can be unreliable when extracting information about the environment and the objects in it to greatly increase the safety of an autonomous driving system. Most importantly depth information can be partly missing or inaccurate, as the extraction of 3D information from multiple 2D images is prone to errors or slight inaccuracies. Without the information of distances between and to objects, the planning of a safe route for the vehicle is not safely computable [27]. To accumulate sufficient 3D information about the environment and the objects in it, an additional sensor, a so-called LiDAR sensor, can be utilized. LiDAR stands for Light Detection And Ranging and describes a method of range detection based on emitting laser light and the measurement of the time it takes until the sensor observes the reflection. With such a LiDAR sensor a point cloud of the environment can be created within a very short period of time and with greater precision than RADAR. This point cloud then contributes accurate 3D information of the environment to the 2D information of the cameras.

Although there are models that achieve promising results, there is still a lot of room for improvement for a 3D object detector to be safe to use. To achieve the required safety a big, diverse and high-quality annotated dataset of 3D point clouds is needed for a Deep Neural Network to be trained on it. The performance of the 3D object detector is strongly dependent on the dataset it was trained on. This is one of the current limitations in this field, as the availability of such datasets is sparse. Additionally these datasets are very expensive and laborious to create, as first of all a car, which is used for capturing the data, has to be modified and equipped with expensive tech. This includes tech such as cameras, RADAR sensors and most importantly a LIDAR sensor, which is used for the point cloud generation. Then someone has to drive the car for an extensive amount of time and try to capture as much distinct data as possible. But only capturing the data is not enough, as it is necessary to annotate the objects to be able to train on this dataset. This annotation process is the most laborious and therefore most time consuming process of the dataset generation, as it is very difficult to accurately label objects in three-dimensional space. All these reasons contribute to the incentive to find a way to synthetically create 3D object detection datasets. Synthetic data, is data that is created by a computer

[1] https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries, Access time: 25 May 2021
[2] https://www.sae.org/standards/content/j3016_201806/, Access time: 25 May 2021
[3] https://www.tesla.com/de_AT/autopilot, Access time: 25 May 2021

simulation or more specifically data taken from engines like game engines or Virtual Reality (VR) engines.
The usage of such game engines for the dataset generation eliminates all the reasons, which make the generation of a 3D object detection dataset so expensive and time consuming. First there is no real car needed, which has to be modified and equipped with expensive tech, as the LiDAR can be simulated within the engine. Then the capturing process can be automated and with that the annotation process can be included as well, as the ground truths of all objects in the frame are known at any point and can therefore be saved without manual effort. Another advantage of the usage of such engines is the possibility of simulating different specifications for the capturing device just by changing parameters like resolution, range, Field of View (FoV) or its position. With that the dataset generation for different types of sensors is fast and uncomplicated. The generation of the point clouds is usually done by simulating the LiDAR within the engine via ray-casting. Ray-casting describes the process of emitting virtual rays to receive 3D information about the environment.

The challenge with synthetic data is, that there are differences between the real and synthetic data called domain gap. These differences lead to bad detection scores when evaluating real world data on a 3D object detector which was trained on synthetic data [21]. Reasons for the domain gap are differences between the point cloud capturing process in the synthetic world and the real world. In the real world the point cloud usually is degraded by outliers and noise. Factors for this degradation can be the lighting or reflective quality of the scanned object, which cannot yet be simulated to the same degree in the virtual world [21, 29]. This is due to the complex task of simulating the emitted laser light by the LiDAR in combination with the return of the reflection of this emitted light, as the reflection is dependent on the material of the hit object. Different materials reflect and absorb light differently and this specific simulation of different materials is not yet fully realizable in current simulators or game engines. This leads to degradations that are present in point clouds captured in the real world but are missing in the synthetic point clouds, which leads to bad detection scores when evaluating models trained on synthetic data on real world data.
But the domain gap is not only present when evaluating 3D object detectors trained on synthetic data on real world data. Also when evaluating a 3D object detector trained on data from a specific region can lead to bad detection scores if it is evaluated on a dataset from a different region. This, for example can relate to different models of cars. This was shown by Wang et al. [24] as they evaluated a 3D object detector trained on the KITTI dataset [3], which was fully recorded in Germany, on 4 different datasets recorded in the United States of America. They found that the differences in car sizes lead to a significant dump in accuracy, as cars in the USA are in general bigger than in Germany. This confirms the fact that deep neural networks are keen to overfitting, which means that they learn dataset specific properties, like for example car sizes. Additionally the used LiDAR properties for the dataset generation itself play a significant role in the performance of the 3D object detector and can lead to a domain gap. The reason for that can be the differences between the LiDAR properties used for the generation of the dataset and the LiDAR properties of the target domain. As differences in the properties of the LiDAR, such as resolution, lead to different representations of the environment which in turn contributes to the domain gap. This is, as already mentioned, where synthetic datasets are advantageous, as different specifications for the LiDAR can be simulated without much effort.

For the generation of synthetic datasets three different approaches emerged. The first one being the generation of synthetic datasets for 3D object detection by using computer games like Grand Theft Auto V (GTA V)[4] [16]. The reason why video games are used for the synthetic dataset generation are the realistic worlds presented in games like GTA V. This relates to graph-

[4] https://www.rockstargames.com/de/games/V, Access time: 25 May 2021

ics, world design and most importantly the population of the virtual world with diverse types of interactive objects like cars, pedestrians or cyclists. With that video games usually provide virtual worlds, which are very close to the real world.
Although video games like GTA V have stunning graphics and realistically populated virtual worlds, the objects itself are often not represented accurately in the virtual world. For example in GTA V pedestrians are reduced to simpler objects such as cylinders to make the representation of the object easier for the graphics engine [16, 25]. This can lead to incorrect point clouds and therefore to unusable data.

Other than the approaches of synthetic data generation with video games, different types of specifically created simulators like CARLA [1] are used to generate synthetic datasets. These autonomous driving environments are created with commercial game engines like Unreal Engine 4[5] or Unity 3D[6] to deliver photo-realistic virtual worlds made for autonomous driving related tasks. These game engines tackle the problem of the incorrect or faulty representation of objects as it is the case with the previously discussed video games. But simulators usually don't represent the a virtual world as close to the real world as video games do, due to the lack of personal and financial resources needed to build a diverse and realistically populated virtual world.

The last approach to synthetic 3D LiDAR data generation is with synthetic datasets which supply depth images, such as the Apollo Synthetic dataset [7] we opted to use. The Apollo Synthetic dataset contains over 270k distinct frames from the Apollo Game Engine Based Simulator[7] and includes different environments like highway, residential or downtown. Additionally in the Apollo Synthetic dataset pedestrians and other objects are modeled correctly and are not reduced to simpler geometric shapes as it can be the case with video games. Alongside the depth images the dataset provides multiple types of ground truth and image data for each frame. The depth images enable the calculation of a LiDAR specific point cloud. It is possible to generate different datasets for different LiDAR's by only having to adjust specific properties like for example the resolution. This possibility of LiDAR specific dataset generation could close the domain gap that appears when using a 3D object detector on a dataset which was created with a different LiDAR specification than the one used in the training process. In our work we first show how to simulate the virtual LiDAR and thus show how to create a LiDAR specific point cloud from a depth image. With this we create a synthetic point cloud dataset from the Apollo Synthetic dataset and train a PointPillars [10] 3D object detection model on our synthetic dataset. To be able to draw a conclusion from the performance of our model we conduct different experiments by evaluating our 3D object detector on the real world KITTI dataset [3]. Further we show the impact the correct simulation of a LiDAR has on the performance of the 3D object detector, as well as how the introduction of features like noise can improve the performance.

2 Related work

The availability of real world point cloud datasets for 3D object detection is sparse and these datasets are specific to a certain LiDAR and a certain location. Therefore the generation of synthetic datasets has become a well-studied research area, as they can be created for different locations and for different LiDAR's without much effort. This section discusses different approaches for generating synthetic point clouds from different sources. Further this section gives an overview on different 3D Object Detection Networks and also touches on the area of domain adaption.

[5] https://www.unrealengine.com/en-US/, Access time: 25 May 2021
[6] https://unity.com/, Access time: 25 May 2021
[7] https://apollo.auto/gamesim.html, Access time: 25 May 2021

2.1 Synthetic Datasets

Richter et al. [16] showed that the computer game GTA V can be used to create pixel-accurate semantic ground truths. By accessing the communication between the GPU and the game they were able to acquire associations between resources and their corresponding pixels in the frames. With that they automatically dissected the image into areas of pixels which are similar in the combination of texture, shader and mesh. As the combinations of texture, shader and mesh are specific to objects, these areas can then be automatically combined to create pixel-accurate segmentations of the objects in the frame. Further the association of types of objects to combinations of texture, shader and mesh is used to propagate labels. With the propagation of labels the human annotator only has to label a small amount of unlabeld areas which results in a significant speed up in the annotation process. This propagation of labels enabled Richter et al. to create a synthetic dataset consisting of 25 thousand images with the corresponding pixel-wise semantic segmentation data in just 49 hours. This is a significant speed-up compared to similar datasets.
The problem that occurs when using this method is the fact, that video games like GTA V often reduce objects to simpler geometric shapes. This leads to inaccurate representations of models and therefore to inaccurate data. An example for this reduction of models to simpler shapes is the previously mentioned reduction of pedestrians to cylinders which is stated by Wu et al. [25].

Hurl et al. [8] show, that with their approach of extracting synthetic data from GTA V they can create a dataset, which improves the performance of a KITTI [4] based 3D object detection network significantly, if the network is pre-trained with the synthetic data. They provide a dataset consisting of more than 50.000 frames including color and depth image, instance and semantic segmentation, point clouds and labels for every object. The difference between Hurl et al. and other GTA V based synthetic point cloud generations is, that they use a more sophisticated way of simulating the collisions of the simulated LiDAR rays with the objects. Other existing work generate the point clouds by using the in-game ray casting function which partly portrays objects in very simple shapes. Hurl et al. address this problem by not using the in-game ray casting function, but by utilizing the depth buffer, which holds depth information for every pixel of the frame and by converting its values to real depth values. By utilizing these depth values an accurate LiDAR specific point cloud can be generated.
In general video games come with a lot of problems when trying to extract synthetic data from them. First, for a lot of games there are legal restrictions when it comes to extracting data from them. Additionally for a lot of extraction techniques the usage of so-called mods is required to acquire the needed data. These mods can be laborius to setup and the extraction process of data in general can be a cumbersome effort.

As mentioned earlier the same process of generating synthetic LiDAR point clouds can be based on specifically created simulators. Wang et al. [23] use a simulator for autonomous driving research called CARLA [1] to extract LiDAR point clouds. The point cloud extraction process is similar to the GTA V based approach discussed previously as they also equip a virtual car with a LiDAR sensor which then collects data autonomously. To generate the point cloud the collision points of the virtual LiDAR sensor rays are calculated by using the provided ray casting API of the engine. With that Wang et al. create a diverse synthetic dataset which bases on the configurations of the KITTI dataset [3]. Although Wang et al. build a synthetic dataset with the CARLA simulator these simulators are not built for generating synthetic datasets. This is due to the fact that with these simulators everything has to be setup manually. This includes the environment as well as the objects in it. The reason for this being the fact, that these simulators are made to test the performance of an autonomous car in certain situations. Therefore creating synthetic datasets with simulators like CARLA is extremly labourious and time consuming.

Another synthetic dataset worth mentioning is the dataset introduced by Gaidon et al. [2] called Virtual KITTI dataset or also often refered to as vKITTI. This dataset is a virtual clone to the real world KITTI dataset and is created by a real-to-virtual world cloning method. Gaidon et al. first initialize the virtual world with data taken from the KITTI dataset, before cloning the dataset in a semi-automatic manner. Objects like cars are cloned automatically, as there positions in the frame are defined in the real world dataset. The manual effort in the cloning process is restricted to the placing of miscellaneous objects like trees or buildings, where no positional information is available. After the cloning process the virual counterpart to the real world dataset is created. An advantage of the virtual dataset opposed to the real world dataset is the possibility of altering the data, by introducing different weather conditions, adding additional objects or rotating the camera.

The Apollo Synthetic dataset [7], which we utilize to create our synthetic dataset was briefly introduced in the introduction and will be discussed in more detail in the following.
The dataset contains 273.000 distinct frames from various different virtual environments like urban, residential, downtown and highway. These virtual environments were created using the Unity 3D[8] game engine, which is capable of creating highly detailed virtual scenes. From these virtual scenes an accumulation of different data can be extracted for every frame and therefore the Apollo Synthetic dataset provides RGB, depth and segmentation masks for each frame alongside ground truth data for the objects, the lanes and the camera pose.
In the Apollo Synthetic dataset dynamic variations are provided concerning time of day, weather conditions and road degradation. The available time variations are split up into seven different times of day ranging from midnight to evening. The specific times are 12am, 6am, 9am, 1pm, 2pm, 5pm and 6pm. For the weather three different variations are available, which are split up into *clear sky*, *light rain* and *heavy rain*. The road degradation is also split up into three categories, ranging from no degradation to severe degradation. For all these different variations of time, weather and road degradation data for the specific environments is available. These environments are split up into *Downtown*, *Residential*, *Road_Loop_with_Intersections*, *Urban_Intersection* and *Urban_Straight_Road*. Additionally one can choose if the data should include Pedestrians and Traffic Barriers.

2.2 Domain Adaption

As the previously discussed synthetic datasets all suffer from the already mentioned domain gap, a research area evolved which focuses on the minimization of this domain gap. The domain gap appears, when evaluating detection models trained on the synthetic data on real world data. This is due to the differences of the synthetic world to the real world. Therefore the so-called domain adaption focuses on adapting a model trained on synthetic data to the real world data. The evolved research of domain adaption mainly focuses on Unsupervised Domain Adaption, which describes the process of adapting from a labeled source domain to an unlabeled target domain [30].

An interesting approach to unsupervised domain adaption is the Generative Adversarial Network (GAN) [5] introduced by Goodfellow et al. A GAN consists of two networks. The generative network, which tries to generate features regarding to the target domain and the disciminative network, which evaluates these features. The target for the generative network is to learn to generate features, which are so similar to the features of the target domain so that the discriminative network cannot distinguish them from features of the target domain. The discriminative network on the other hand tries to learn to distinguish generated features from real features. Therefore the two networks train each other, which results in generated features closely resem-

[8] https://unity.com/, Access time: 25 May 2021

bling the features of the target domain. Saleh et al. [17] utilize such a GAN to perform domain adaption on synthetic point cloud data.

A different approach to domain adaption is the by Long et al. [12] introduced Deep Adaption Network (DAN). It utilizes a method called maximum mean discrepancy (MMD) to measure the discrepancy between domains. To achieve domain adaption a Deep Neural Network can then be trained to minimize this discrepancy between the domains. An approach using a DAN was introduced by Qin et al. [15] called PointDAN, which performs Domain Adaption on point cloud data.

Other than the GAN and DAN approaches, Su et al. [21] proposed Conditional Domain Normalization (CDN) which extracts domain attributes from the semantic features and uses them to encode the features of other domains. These domain attributes are often neglected in the GAN and DAN approaches.

2.3 3D Object Detection Networks

All models that are discussed in the following are specifically created for point clouds as input, as standard deep neural network approaches cannot extract features automatically from the sparse and extraordinary data that are point clouds [31]. The models can be split into three main groups based on the way of processing the point clouds. The voxel-based models, the point-based models and the models which combine both the voxel- and point-based approach.

2.3.1 Voxel-based networks

Voxel-based networks divide the input point cloud into so-called voxels to simplify the detection process. Voxels are volumetric elements like cuboids or pillars and are used to partition the point cloud into a grid, which can be processed with conventional convolutional neural networks.
VoxelNet [31] is a voxel-based 3D detection network and was proposed in late 2017 by Zhou and Tuzel. The architecture of the VoxelNet is split up into three parts. The first part is the Feature Learning Network which splits the input point cloud up into voxels, groups the points in each voxel, creates voxel-wise features and finally transforms the list of voxel features to a 4D tensor to reduce computation cost and memory usage. These 4D tensors then are processed by the Convolutional Middle Layer, which represents the second part of the VoxelNet architecture and is used to collect spatial associations. The last part of the VoxelNet architecture is the Region Proposal Network which finally generates the 3D detection.

Another voxel-based network is the by Yan et al. [26] proposed Sparsely Embedded Convolutional Detection (SECOND) network which builds upon the VoxelNet and addresses its weakness of high computational cost. This computational cost is decreased by integrating a sparse convolutional layer instead of the standard convolutional layer of the VoxelNet.

PointPillars [10] is an additional example for voxel-based networks, which is widely used. It splits the point cloud into so-called pillars and with that transforms the point cloud to a bird-view grid. On this grid then a 2D convolutional neural network is applied to generate the 3D object detection.

2.3.2 Point-based networks

Point-based networks do not divide the input point cloud as the voxel-based networks do, but rather process the whole point cloud in a point by point manner. This means these methods generate point-wise predictions. As Qi et al. [14] state in their introduction of PointNet, it is important to consider, that the processing of the points should be invariant to the order in which these points are processed and invariant to transformations of the point cloud. PointNet is one of

the fundamental point-based networks and is often used in other approaches [31] for generating point-wise features. It processes the point cloud point by point in an unordered manner and generates point-wise features, from which global features can be created.
PointRCNN [19] proposed by Shi et al. is a good example for a point-based 3D detection network. It conducts 3D object detection in two stages, by first generating 3D proposals for objects from the foreground points of the point cloud data, before refining these proposals and transforming them into canonical coordinates to receive the bounding boxes of the detected objects.

A different example for a point-based network is the Point-based 3D Single Stage Object Detector (3DSSD) introduced by Yang et al. [28] which is, different to the previously discussed PointRCNN, a single stage object detector. This is of importance, as single stage object detectors achieve far better inference speeds than the two stage point-based methods. This is due to the second stage of the two stage methods being a prediction refinement module, which boosts the performance of the two stage networks but leads to slow inference speeds. The 3DSSD network is composed of three parts. The first part is the backbone network, which is used for the global feature generation. These global features are then extracted in the candidate generation layer for characterisic points. Last is the prediction head, which generates the 3D object prediction.

2.3.3 Combination of voxel and point-based features

Next to the voxel-based and the point-based 3D detection networks there is the PointVoxel-RCNN (PV-RCNN) [18], which combines the advantages of both the point-based and the voxel-based networks. The network splits the point cloud into voxels, generates features for each voxel from the point-wise features in each voxel, transforms the voxel features to a bird-view feature grid and from that creates 3D object proposals. Additionally the voxel features are condensed into keypoints. Conclusively these keypoints are used to learn features, which are specific to the generated proposals. These proposal specific features are then used to refine the proposals before generating the final prediction.

3 Approach

This section discusses the generation of the point clouds from the depth images of the Apollo Synthetic dataset [7]. Therefore we will give an overview of the data provided by the Apollo Synthetic dataset, before introducing the point cloud generation process.

3.1 Apollo Synthetic Dataset

As mentioned earlier the dataset provides RGB, depth and segmentation masks for each frame alongside ground truth data for the objects, the lanes and the extrinsic/intrinsic camera parameters. In the following all parts of the dataset will be explained as the subsequent chapter builds on the properties of the data.

Before we discuss the different image types, it is important to note that all images are rendered with a resolution of 1920 * 1080 pixels. The vertical Field of View (FoV) is defined to 30° and the horizontal FoV is 54.8°. Furthermore the intrinsic camera parameters are given as the matrix K in the same notation, as defined by Hartley and Zisserman [6], which looks as follows:

$$K = \begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix} = \begin{bmatrix} 2015 & & 960 \\ & 2015 & 540 \\ & & 1 \end{bmatrix}, \tag{1}$$

where f describes the focal distance of the camera to the plane and is defined as 2015. Variables p_x and p_y define the principal point, which here lies in the center of the image at [960, 540].

The RGB images are color images in the common jpg format. The RGB images can be used for 2D object detection and are therefore often utilized in addition to 3D object detection. As we only focus on 3D object detection we only use them for visualization purposes. An example of a RGB image of the dataset can be seen in Figure 1.

Figure 1: Example for a RGB image of the Apollo Synthetic dataset.

The segmentation masks are given in the png file format and contain semantic and instance-level segmentation per pixel. For all segmentation masks a color encoding text file is provided which contains information about the category-wise segmentation of the image. The corresponding segmentation mask of the RGB image from Figure 1 can be seen in Figure 2.

Figure 2: The corresponding segmentation mask to the RGB image of Figure 1 from the Apollo Synthetic dataset.

The depth images are, as the segmentation masks, given in the png file format, which is a loss-free image datatype. These png images contain 16-bit depth information in the red and

green channel of the image. An example of the depth image corresponding to the RGB image from Figure 1 can be seen in Figure 3. Furthermore the separation of the image from Figure 3 into its red and green channel is presented in Figure 4.

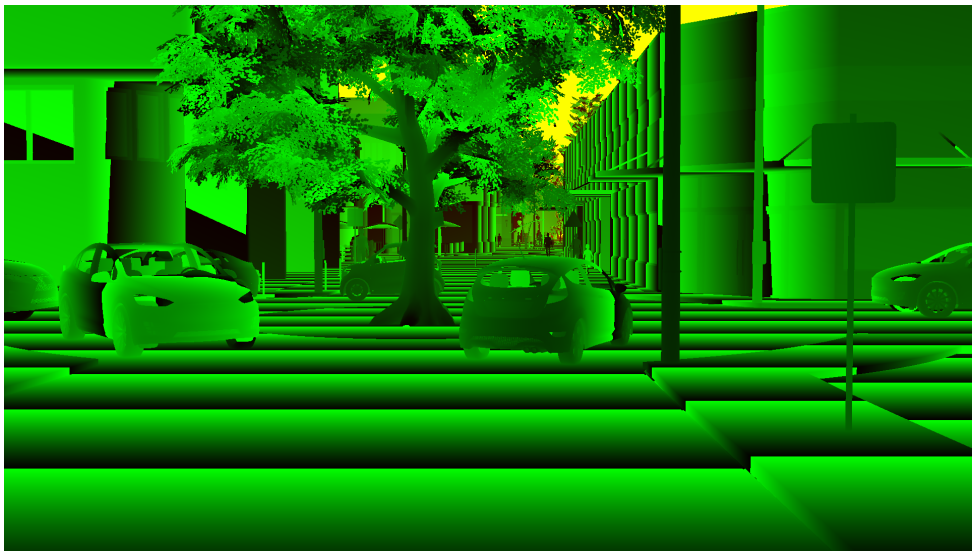


Figure 3: The corresponding depth image to the RGB image of Figure 1 and segmentation mask of Figure 2 from the Apollo Synthetic dataset.

(a) red channel (b) green channel

Figure 4: This figure shows the red and green channel for the depth image shown in Figure 3 from the Apollo Synthetic dataset.

To be able to achieve a precision of 1 cm the maximum distance in the depth image is fixed to 65535 cm. This allows a normalization of the depth value to the 16 bit integer range of [0; $2^{16} - 1$] or in a better representation [0; 65535] and thus 1 cm precision is achieved. For the decoding of the depth values from the red and green channels of the depth images the Apollo Synthetic datasets description provides an Equation

$$Z = (R + \frac{G}{255.0}) * 65536.0, \tag{2}$$

which has to be applied for every pixel of the depth image. The variables R and G denote the values of the Red and Green channel of the depth image normalized to the range [0.0;1.0].

Next to the just described image types, the Apollo Synthetic dataset also provides three different types of ground truth data, which will be introduced in the following.
The first ground truth data is the object ground truth data, which holds positional information

for every object. It is provided in form of a text file with each object filling one line in the ground truth file. The values, regarding the positional information of the objects, are split by spaces, which simplifies the importation process of the ground truth data in the programming environment. The information saved in these object ground truth files includes the frame and object id, the object category, flags for truncation and occlusion, values for the 2D bounding box, as well as values describing the object size in combination with the object center and its rotation both in camera and world coordinates. It is worth mentioning, that the object ground truth files include the necessary values to be compatible with the widely used format introduced by Geiger et al. [4]. The two remaining ground truth data are the 3D lane ground truths and the extrinsic camera parameters for every frame. The lane ground truth gives information about the lane markings on the road and as our work specializes on 3D object information we don't use the lane ground truth data. The files containing the extrinsic camera parameters hold the extrinsic matrix of the camera for every frame. The extrinsic matrix consists of the rotation matrix R and the camera translation vector T. As we generate the point clouds file by file, there is no need for the knowledge, about where the car is in the world space and therefore we have no use for the extrinsic camera parameters.

Issues

While working with the Apollo Synthetic dataset we encountered some issues concerning the accuracy of the 3D object bounding boxes. First the bounding boxes of the pedestrians are inaccurate, as they are widely oversized. Second are the undersized bounding boxes of the cars, where most of the time the side mirrors lie outside of the bounding box. The oversized bounding boxes are visible in Figure 5 and Figure 6 shows an example of an undersized bounding box of a car, where the side mirror lies outside of the bounding box.

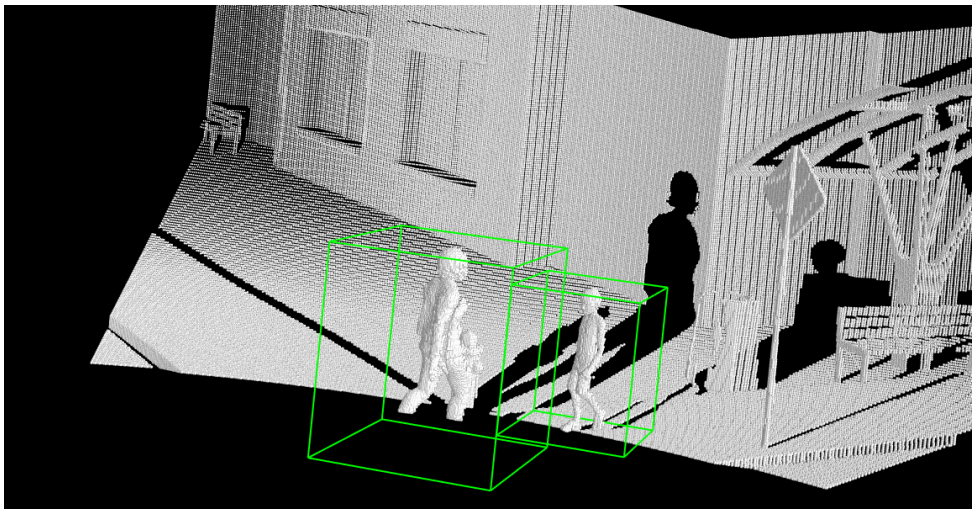


Figure 5: Representation of the oversized bounding boxes for pedestrians in the Apollo Synthetic dataset.

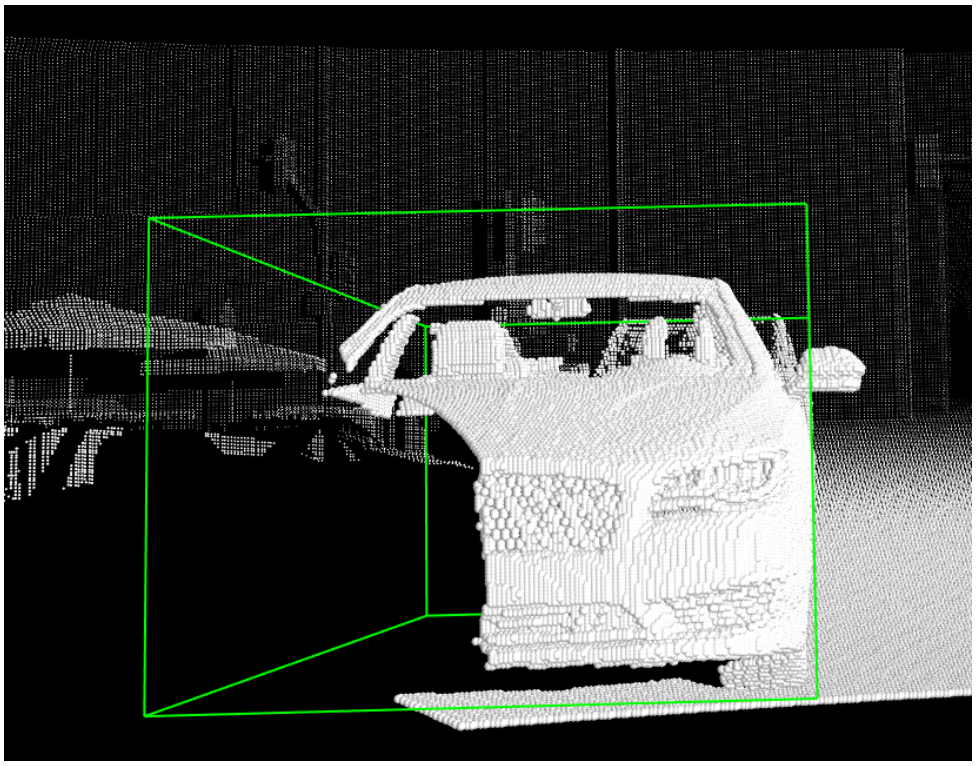


Figure 6: Representation of an undersized bounding box of a car in the Apollo Synthetic dataset.

3.2 Point cloud generation

This chapter discusses the generation of point clouds from the corresponding depth images of the Apollo Synthetic dataset. The presented equations used for the calculation of the 3D coordinates are based on Chapter 6.1 from the book Multiple View Geometry in Computer Vision [6]. As was shown in the last chapter, the depth information is encoded into the red and green channel of the image and can be decoded with Equation 2 in order to get the depth values required for the point cloud generation. To generate all depth values of the depth image, Equation 2 has to be applied for every pixel. After applying the formula to a pixel in the depth image we receive the Z-value for that pixel.

The used camera geometry of the Apollo Synthetic dataset is a simple pinhole camera with a right handed coordinate system. This means, that X is going to the right, Y is going down and Z is going forward, as it represents the depth value. The pinhole model describes the image plane or also called focal plane as a plane placed in front of the camera center with a distance of f. In the Apollo Synthetic dataset the focal distance f is defined as 2015 and can be taken from matrix K, which holds the intrinsic camera parameters. Additionally the principal point $[p_x; p_y]$ is defined in the intrinsic camera parameters as [960; 540]. The principal point defines the point, where a vector originating from the camera center pointing straight towards the image plane intersects this image plane. These intrinsic camera parameters in combination with the depth values we calculate from the depth image enable us to calculate 3D points from the corresponding 2D points. We can do this by utilizing the mapping of the camera space 3D coordinates (X, Y, Z) to the 2D coordinates of the image space ($f * X/Z + p_x$, $f * Y/Z + p_y$). The coordinates of the principal point are specified as p_x and p_y. This leads to the mapping of the camera space coordinates (X, Y, Z) to the image coordinates (x, y), where x and y are given as

$$x = \frac{f * X}{Z} + p_x, \tag{3}$$

$$y = \frac{f * Y}{Z} + p_y. \tag{4}$$

To be able to compute camera space coordinates from image space coordinates we have to transform the mapping to fit our need. As we want to calculate the 3D coordinates from our 2D coordinates of the image we have to transform Equation 3 and Equation 4 in order to calculate X and Y. After transforming the two equations according to [6] we get:

$$X = \frac{(x - p_x) * Z}{f}, \tag{5}$$

$$Y = \frac{(y - p_y) * Z}{f}. \tag{6}$$

As mentioned previously for the Apollo Synthetic dataset the principal point lies at the center of the image, which lays at 960 pixels horizontally from the left edge of the image and 540 pixels vertically from the top of the image. This point represents our center, where both X and Y of the camera coordinates and x and y of the image coordinates are 0. So the principal point is at (0, 0) in image coordinates and at (0, 0, f) in camera coordinates. To calculate the 3D coordinates from the 2D coordinates we have to consider this offset of the principal point in our calculations. Therefore the respective x and y-part of the principal point [p_x;p_y] are included in Equation 5 and Equation 6.

An example of a point cloud, where every pixel of the depth image was transformed into 3D coordinates can be seen in Figure 7. For reference the corresponding RGB and depth image are presented in Figure 1 and Figure 3 respectively.

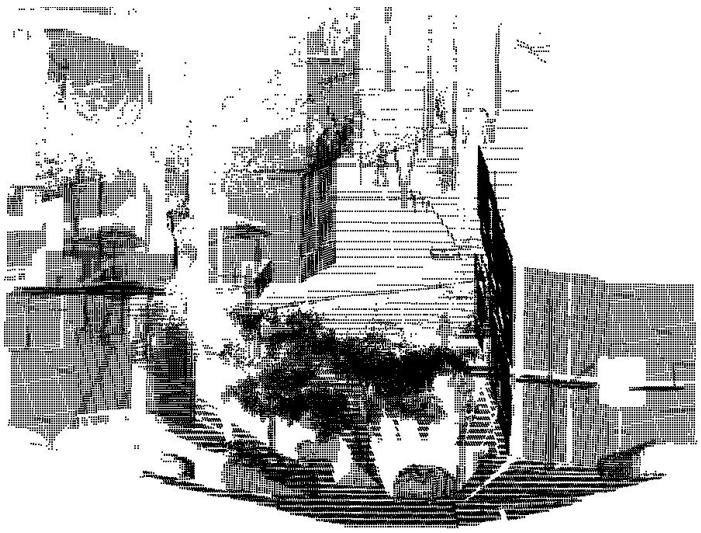


Figure 7: Corresponding point cloud to the depth image of Figure 3 from the Apollo Synthetic dataset. The viewport differs from the corresponding images due to the point cloud viewer.

As we want to simulate a LiDAR for the creation of the point clouds, we have to place a virtual LiDAR at the camera center. As stated in the introduction, LiDAR stands for Light Detection And Ranging and it is used for range detection. This range detection bases on the emission of laser light and the time passed until the sensor observes the reflection. Typically the sensor emitts the laser ray while rotating and thus reaches a horizontal angle range of up to 360°. The vertical resolution depends on the vertical FoV of the LiDAR in combination with the

amount of channels. A channel describes one vertical position of the sensor. As an example a LiDAR with a vertical FoV of 33.2° and 16 channels emits rays with a vertical spacing of 2.075°. A representation of this example can be seen in Figure 8.

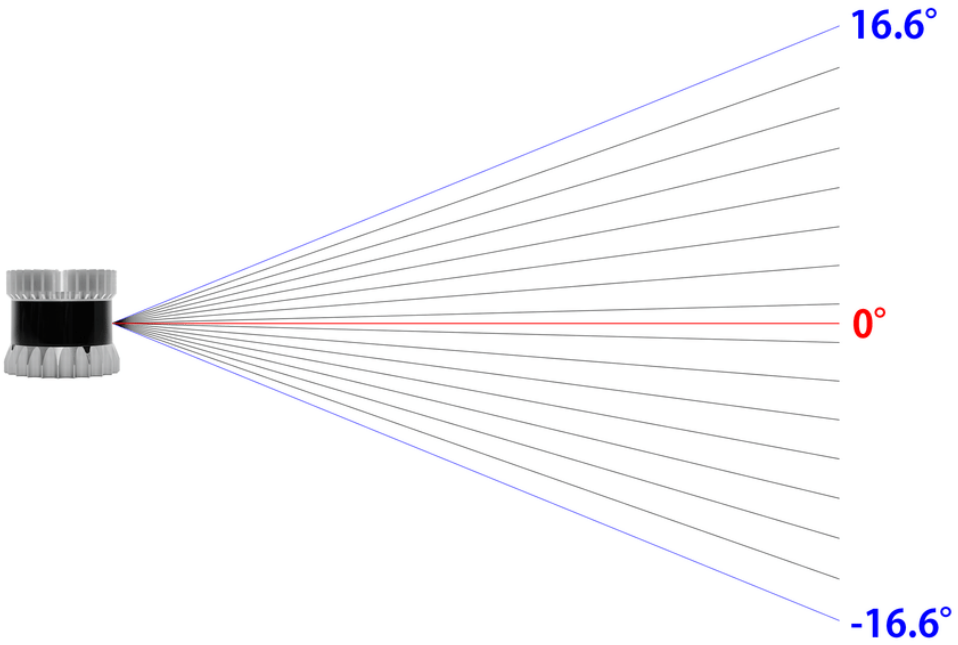


Figure 8: Representation of a 16-channel LiDAR with a vertical FoV of 33.2°. Image taken from [9].

The virtual LiDAR in the camera center then needs to be simulated accordingly. This means we need to simulate the emission of the laser light by the LiDAR and the intersection of these rays with the environment. This is done by calculating every point on the image plane which is hit by the respective LiDAR ray. As the laser of the LiDAR is moving in a circular motion, the points on the image plane, which are hit by the rays, have to be calculated considering this circular motion. Due to the rotation of the LiDAR, each emitted ray creates a cone, as represented in Figure 9. The intersection of this cone with the image plane creates a parabola on the image plane rather than a horizontal line, which would be the result when the circular motion of the LiDAR is not taken into account.

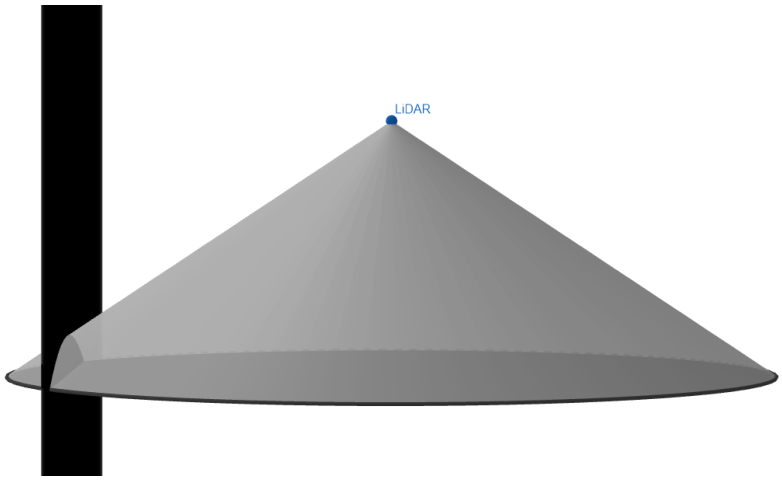


Figure 9: Example for a cone resulting from the emitted ray in combination with the roation of the LiDAR. The intersection of the cone with the image plane (black plane) results in a parabola.

To now consider the circular motion of the LiDAR we have to utilize sperical coordinates for the correct simulation of the rays emitted by the LiDAR. Therefore we calculate the respective Cartesian coordinates (x, y, z) by using the spherical parameters r, θ and φ. The Radius r is set to 1 because we only need to know the direction the vector points towards, as this vector represents an emitted ray of the LiDAR. With this vector we can calculate the point on the image plane this ray intersects. Parameters θ and φ are the angles used to describe a point in spherical coordinates, where θ represents the angle between the y–axis and the z–axis and is also called the polar angle. The angle φ is also called the azimuthal angle and represents in our approach the angle between the z-axis and the x-axis. This is due to the interchange of the z and y-axis in our approach. Figure 10 visualizes the normal spherical coordinate system. By the interchange of the z and y-axis we receive our representation of the spherical coordinates in which the y-axis points up, the z-axis points to the front and the x-axis points to the right. This means the z-axis points towards the principal point on the image plane and a ray with polar and azimuthal angle of 0° should follow this z-axis. Additionally, by interchanging the z and y-axis we can see, that the angles θ and φ both describe angles between the z-axis to one of the other two axes. Therefore we initially have to set both angles to 90° to get a vector which points towards the z-axis and therefore towards the principal point of the image plane. This means we have to add 90° to every angle we want to simulate with the LiDAR as a ray with 0° polar angle and 0° azimuthal angle should point towards the principal point of the image plane.

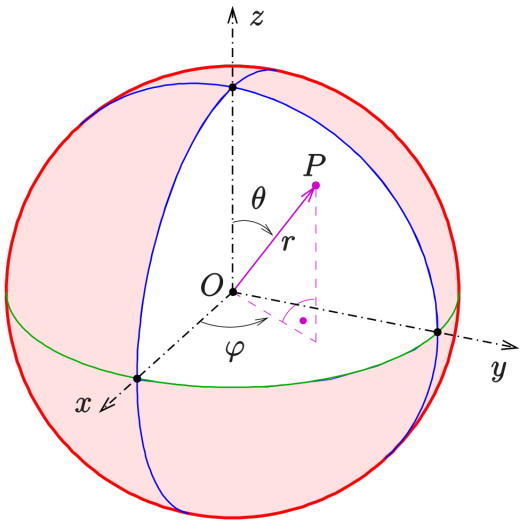


Figure 10: Standard spherical coordinate system. Image taken from [9].

To calculate the Cartesian coordinates of all vectors, which represent emitted rays by the LiDAR, we need to calculate all horizontal and vertical angles of the emitted rays. This means we need to calculate the polar and azimuthal angle for every emitted ray. To be able to do this we need to know the horizontal and vertical resolution of the LiDAR. This means we need to know the horizontal angle between two consecutive rays and the vertical angle between two consecutive channels. This horizontal and vertical resolution are parameters which are specific to the LiDAR and therefore given parameters depending on the used LiDAR. Further LiDAR-specific parameters are the vertical and horizontal Field of View (FoV). This vertical and horizontal FoV needs to be split accordingly to fit our representation of the coordinate system with the z-axis pointing towards the center of the image plane. Therefore both the horizontal and the vertical FoV need to be split in half to fit the range $[-\frac{FoV}{2};\frac{FoV}{2}]$. With the horizontal and vertical resolution, in addition to the defined FoV range, it is possible to calculate the angles θ and φ for every ray. The angle according to the horizontal resolution is represented by θ and φ is the angle representing the vertical resolution.

After calculating the angles for an emitted ray of the LiDAR we can calculate the corresponding Cartesian coordinates for the emitted ray. Figure 10 visualizes this with the point P, given in Cartesian coordinates. The vector relating to point P represents an emitted ray and is usually denoted as L. To calculate the Cartesian coordinates with the parameters of the spherical coordinates we only have to insert the radius r and the two angles θ and φ into these three equations:

$$x = r * sin(\theta) * cos(\varphi), \tag{7}$$

$$y = r * cos(\theta), \tag{8}$$

$$z = r * sin(\theta) * sin(\varphi). \tag{9}$$

These are the equations needed for calculating Cartesian coordinates from spherical coordinates, with the modification that z and y are interchanged due to the transposition of the y and z-axis in our approach. Parameters θ and φ are the vertical and horizontal angles we calculated

[9] https://commons.wikimedia.org/w/index.php?curid=41627945, Access time: 25 May 2021

for the emitted ray plus the already mentioned 90° to initially start at the z-axis pointing towards the center of the image plane. With that we can calculate the vector which represents the emitted ray, which we then intersect with the image plane.

To calculate the intersection point of the vector with the image plane we implement the equations of the line-plane intersection. (Information on line-plane intersection was taken from [10].) A plane in vector notation can be represented as a set of points p for which the following condition stands:

$$(\mathbf{p}–\mathbf{p_0}) * \mathbf{n} = 0. \tag{10}$$

In this context n is the normal vector to the plane and p_0 is a point on the specific plane. Further a line is represented by Equation

$$\mathbf{p} = \mathbf{l_0} + \mathbf{L} * d, \tag{11}$$

where L is a vector representing the ray to be intersected, l_0 is a point on this ray and d is a real number scalar. By combining Equations 10 and 11 we get Equation

$$((\mathbf{l_0} + \mathbf{L} * d)–\mathbf{p_0}) * \mathbf{n} = 0. \tag{12}$$

This can be expanded to Equation

$$(\mathbf{L} * \mathbf{n}) * d + (\mathbf{l_0}–\mathbf{p_0}) * \mathbf{n} = 0 \tag{13}$$

and solved for d to result in Equation

$$d = \frac{(\mathbf{p_0}–\mathbf{l_0}) * \mathbf{n}}{(\mathbf{L} * \mathbf{n})}. \tag{14}$$

The condition for an intersection by the line with the plane is, that $L * n$ is not equal to 0. So when $L * n \neq 0$ holds, the point of intersection can be calculated with Equation 11.
Before we can calculate if a vector intersects the plane, we have to define some values. First we define the normal vector n, which has the values (0, 0, 1) and is therefore normal to the image plane. Then we have to define the point p_0 which has the values (0, 0, f) and thus lies on the image plane. More specifically p_0 lies at the center of the image plane, as (0, 0) points towards the center and f is the distance to the image plane. The last value we have to define is l_0, which should be a point on every line we want to check for intersection. Therefore we choose l_0 to be (0, 0, 0) as this is the origin of every ray and therefore it is guaranteed to be on the line.

To now calculate the line-plane intersection for the simulated ray of the LiDAR, we have to utilize two equations. First we need to calculate d, which can be done with Equation 14. Additionally to the defined constants p_0, l_0 and n we need to insert the vector L into the equation. Vector L represents an emitted ray of the LiDAR and relates to point P as shown in Figure 10. The value d we receive from Equation 14 then can be used in Equation 11, to calculate the point p. The point p represents the point, where the intersection of the ray with the image plane happens. As the intersection point p is given in image coordinates, where the principal point lies at (0,0), we need to add the offset of the principal point $[p_x;p_y]$ to point p. With the addition of the offsets p_x and p_y we receive the position of the pixel on the depth image, where the ray hit the image plane. This pixel then holds the depth information we need to calculate the 3D coordinates of the corresponding point in the point cloud. It is important to notice, that it is possible for rays to hit the image plane outside of the defined area. This can happen if the LiDAR has a greater FoV than the image used for the intersection of the rays. Therefore a check is required to make sure the point of intersection lies on the image.

[10] https://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection, Access time: 25 May 2021

The just described calculation of the intersection of a ray with the image plane has to be done for every emitted ray. As the calculations for the intersection of the rays with the image plane are independent from each other matrices can be used to speed up the calculation process.

As an example Figure 11 shows a point cloud, which was created by simulating a Velodyne HDL-64E LiDAR and intersecting the rays with the same depth image of the Apollo Synthetic dataset shown in Figure 3. The Velodyne HDL-64E LiDAR sensor has a horizontal FoV of 360° and a vertical FoV of 26.8°. It emitts 64 beams with a horizontal resolution of 0.09° and a vertical resolution of ∼0.42°. The maximum range of the Velodyne HDL-64E LiDAR is 120 meters and therefore points with a greater distance are not displayed.

Figure 11: This figure shows the resulting point cloud based on the simulation of the Velodyne HDL-64E LiDAR on a depth image of the Apollo Synthetic dataset.

Additionally to the point cloud shown in Figure 11, Figure 12 shows the intersected points on the image plane. It nicely displays the parabolas, which are the result of the intersection of the created cones and the image plane, as was shown in Figure 9.

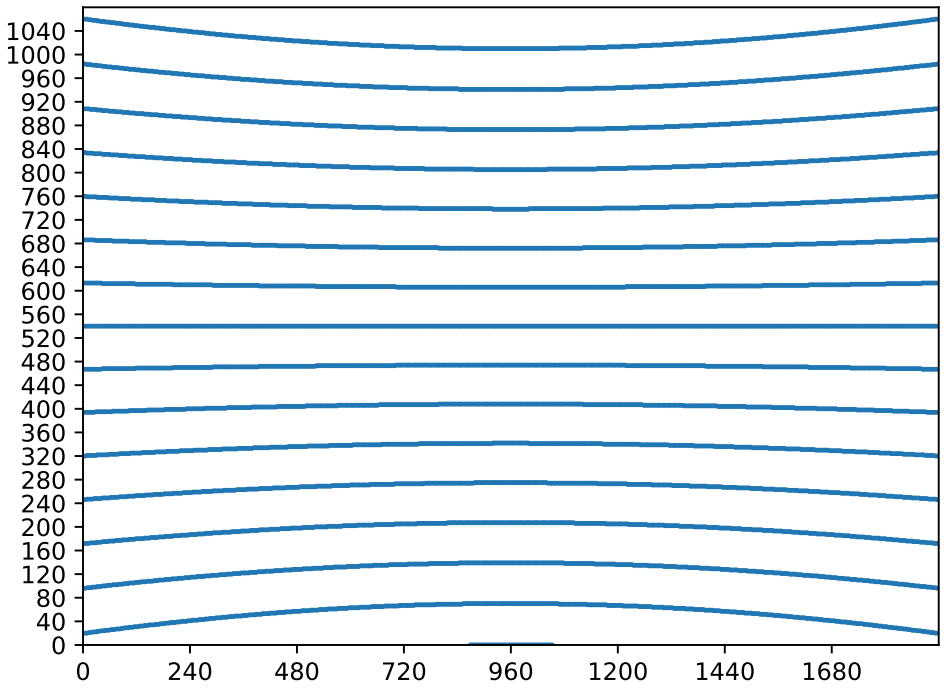


Figure 12: This figure shows the points on the image plane, where the LiDAR beams intersect the image plane.

3.3 3D Object Detection

The model we use to train a 3D object detector is the by Lang et al. [10] introduced PointPillars model. We decided to use the PointPillars model as it is a state of the art approach with high computation efficiency leading to fast training times. Additionally PointPillars is one of the supported models of the OpenPCDet framework [22] we utilize for the training process.
As stated in the related work section, the PointPillars model falls into the category of the voxel-based models. The PointPillars network can be split up into three parts, which is shown in Figure 13. The first part is the Pillar Feature Net, which generates a 2D pseudo-image from the input point cloud. This is achieved by first partitioning the point cloud into pillars. Pillars are created by splitting the point cloud into a grid from the bird-eye view. Then all points in the point cloud are extended with the values x_c, y_c, z_c, x_p, y_p. Variables x_c, y_c and z_c represent the distance of a specific point inside a pillar to the arithmetic mean of all points in this specific pillar. Variables x_p and y_p describe the point's offset from the center of the pillar from the bird-eye view. With these 5 additional values, one point is converted from a 4-dimensional point (x, y, z, r), with r being the reflectance value, to a 9-dimensional point (x, y, z, r, x_c, y_c, z_c, x_p, y_p). Due to the sparsity of point clouds the pillars will be mainly empty or at least contain a small amount of points. PointPillars exploits this sparsity by implementing a limit N on the number of points per pillar as well as a limit P on the number of non-empty pillars. With this, Lang et al. create a dense tensor of size (D, P, N). If the amount of data a pillar contains exceeds the capacity of this tensor the data is sampled randomly. Inversely zero padding is applied, if the amount of data a pillar contains is too little to populate the tensor. These tensors are then used to learn features by applying a simplified version of PointNet [14]. First a linear layer is applied followed by a batch normalization layer and a ReLU layer before then applying a max-pooling operation to generate encoded features. These encoded features are then scattered back to the original pillar locations to create a 2D pseudo-image. After generating the pseudo-image a 2D convolutional neural network is applied on it to generate features, which are then used by the Single Shot Detector (SSD) [11] to generate the 3D object prediction. [10]

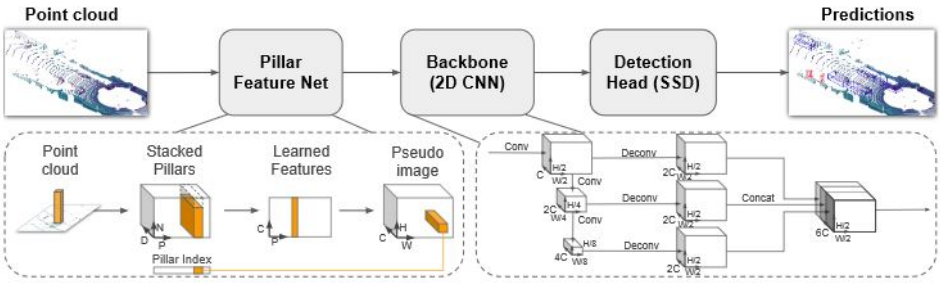


Figure 13: PointPillars network architecture. Image taken from [10].

4 Evaluation

To test the performance of our generated synthetic dataset in comparison to a real world dataset, we trained a 3D object detector on the synthetic dataset and evaluated it on the KITTI dataset [3]. Further we evaluated differently trained 3D object detectors, trained on the synthetic dataset with different specifications, and showed the importance of simulating the LiDAR parameters correctly. The LiDAR configuration we based our experiments on is the used LiDAR from the KITTI dataset, which is the Velodyne HDL-64E. It emits 64 Beams with a maximum range of 120 meters and an accuracy of smaller than 2 cm.
For training the 3D object detector, we utilized the OpenPCDet framework [22] (Version 0.3.0). OpenPCDet provides a PyTorch-based codebase for 3D object detection from point clouds. OpenPCDet supports multiple state-of-the-art 3D object detection networks, such as PointPillars [10]. For convenience OpenPCDet defines all model-specific parameters in config-files, which simplifies the training process of a customized model significantly. Additionally the OpenPCDet framework supports different real world datasets, including the KITTI dataset we used as a real world reference in our experiments.
To train the 3D object detection models we used a desktop PC with an Intel i7 6700k, 16 GB of RAM and a GTX 1070 graphics cards. The used operating system was Ubuntu 20.04 LTS and we trained the models in a Docker container running the OpenPCDet framework.

4.1 Dataset Preparation

As we used the KITTI dataset as a real world reference for our synthetic dataset and as the KITTI dataset is supported by the OpenPCDet framework we had to prepare our dataset to resemble the supported KITTI dataset in order to be able to train on it.
The first adjustment of the synthetic dataset needed to resemble the KITTI dataset is the rearrangement and transformation of the x, y and z-axis. In the coordinate system of the synthetic dataset the x-axis points to the right, the y-axis points down and the z-axis points towards the front direction. The KITTI dataset uses a coordinate system in which the x-axis points towards the front direction, the y-axis points to the left and the z-axis points up. Therefore we needed to rotate our coordinates x,y,z which equals to $right, down, forward$ by one to get the coordinates z,x,y which corresponds to $forward, right, down$. Additionally to the rotation we needed to flip our x and y-axis from $right, down$ to $left, up$. This led to the final coordinate representation z, $x_{flipped}$, $y_{flipped}$ which fits the coordinate representation of the KITTI dataset. The second adjustment we implemented is a limitation to the depth value. The maximum distance for the point clouds is defined to be 70 meters. Therefore we limited the point clouds in the generated dataset to 70 meters and only included objects, which lie within this maximum distance.

4.2 Implementation Details

As the config-file for the PointPillars model, which is provided by the OpenPCDet framework, bases on the KITTI dataset we needed to make some adjustements in order to train an object detector on our dataset. The config-files provided by the OpenPCDet framework are split up into three main parts. The first part is the *Data_Config*, which defines data specific parameters like the point cloud range, voxel size and the data augmentation. The second part in the config-file is the *Model*, where all details of the model and its architecture are defined. The last part is the *Optimization* part, where training-specific parameters like batch size or learning rate are defined.

Important parameters in the *Data_Config* part of the config-file are the aforementioned point cloud range, voxel size and the data augmentation. We left the point cloud range and the voxel size untouched. The point cloud range is defined as $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$ and holds the values [0, -39.68, -3, 69.12, 39.68, 1]. The voxel size is given as $[x, y, z]$ and holds the values [0.16, 0.16, 4]. This means each voxel is a quadratic pillar with dimensions 0.16m x 0.16m x 4m. In regard to the point cloud range and the voxel size it is important to note, that when dividing the axis-specific part of the point cloud range with the corresponding part of the voxel size the result should be an integer. As an example, when dividing the y-part of the point cloud range with the y-part of the voxel size we get: $\frac{79.36}{0.16} = 496$. This relation between the point cloud range and the voxel size is needed to make sure that the full point cloud range can be voxelized. The data augmentation consists of multiple parameters describing different data augmentors. In general we left the data augmentor parameters untouched, except for the USE_ROAD_PLANE parameter of the *gt_sampling* augmentor, which we set to $false$, as we had no road plane data. The PointPillars config-file includes 4 data augmentors. The first data augmentor is the by Yan et al. [26] introduced *gt_sampling* augmentor, which basically extracts the point cloud data for every object in the training dataset. It then builds a database holding the point cloud data for each object. During the training process objects are then randomly selected from this database and introduced in the current point cloud to increase the number of objects per point cloud. By utilizing *gt_sampling* Yan et al. showed that the performance of the model could be increased significantly.
Next to the *gt_sampling* augmentor there are three additional data augmentors defined in the config-file, which augment the point cloud representation by flipping it around a defined axis, rotating it or scaling it. As with the other data augmentor we did not change parameters for these augmentors.

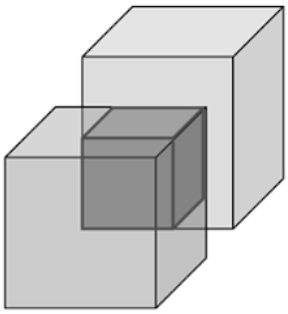
In the *Model* part of the config-file the parameters for the model itself including its architecture are defined. If a dataset different to the KITTI dataset is used for the training process the parameter for the object anchor sizes needs to be adjusted. These anchor sizes are a parameter of the configuration for the dense head, which builds the last part of the PointPillars architecture and is responsible for the generation of the 3D object predictions. The anchor sizes are defined for every supported object class and define the average size of a bounding box for an object from the specific class. As the Apollo Synthetic dataset mainly includes american cars and as the predefined anchor sizes base on the KITTI dataset we needed to adjust these anchor sizes. This was necessary due to the differences in car sizes between american cars and the european cars present in the KITTI dataset, which are part of the domain gap as was shown by Wang et al. [24]. Therefore we calculated the average bounding box size for each object in our dataset and adjusted the defined values in the config-file. Except for the anchor sizes we left the *Model* part of the config-file untouched.

The last part of the config-file is the *Optimization* part. In this part training-specific parameters like batch size, learning rate or the number of epochs are defined. These parameters can be adjusted according to the experiment and the machine used for the training process. We did

not adjust these parameters and trained our models for 80 epochs with a batch size of 4. Errors occuring during the training process according to memory exhaustion are usually related to a too large batch size.

4.3 Evaluation Metric

3D Intersection over Union (IoU) was used as the evaluation metric for the generated bounding boxes. IoU in the application of 3D object detection describes the metric for measuring the overlap of the predicted bounding box with the ground truth bounding box. IoU is calculated by dividing the overlap of the bounding boxes with the union of the bounding boxes. A visual representation for the IoU calculation in 3D is shown in Figure 14.



$$\boldsymbol{3D\ IOU = \frac{Intersection(obj_1, obj_2)}{Union(obj_1, obj_2)}}$$

Figure 14: Representation of the IoU calculation for 3D predictions. Image taken from [20].

The result of the IoU calculation then gives information on how good the prediction is. An IoU of 1 means the prediction matches the ground truth perfectly and an IoU of 0 means the prediction is completely wrong. This result of the IoU calculation is used to determine if a prediction is a true positive or a false positive. According to Padilla et al. [13] a prediction is classified as a true positive if the ground truth bounding box is predicted correctly. This condition if a bounding box is predicted correctly depends on the set threshold for the IoU, as the calculated IoU needs to be greater than the threshold to be counted as a correct prediction. The threshold we used in the experiments was 0.5, meaning the IoU between the predicted and the ground truth bounding box needed to be greater than 0.5 to be counted as a true positive. Predictions which do not reach an IoU of 0.5 or do not predict the right class are counted as false positives. As stated by Padilla et al. with the true positives (TP) and the false positives (FP) the precision of the object detector can be calculated with the Equation

$$precision = \frac{TP}{TP + FP}. \tag{15}$$

4.4 Experiments

Our conducted experiments can be divided into two main categories. The first category discusses the correct simulation of the LiDAR parameters and the impact the correct simulation has on the performance. The second category describes the experiments which focused on the simulation of real world effects and their performance impact.
An important thing to note is, that we only evaluated on cars. This is due to the inaccurate bounding boxes for pedestrians as well a the non-sufficient amount of training data for cyclists.

Current State
In order to draw a conclusion from our experiments in regard to the performance of the models we compared our results to the results presented in the paper published by Su et al. [21]. Su et al. introduced their Conditional Domain Normalization (CDN) model and compared their results to other state of the art approaches, as can be seen in Table 1. *Target* denotes a model trained on the real world KITTI dataset and represents the target accuracy for a 3D object detector trained on synthetic data.

Model	**Car**
CycleGAN [17]	16.5
PointDAN [15]	17.1
CDN [21]	19.0
Target	75.7

Table 1: Performances of state of the art approaches as well as the performance of a 3D object detector trained on the real world KITTI dataset denoted as *Target*.

4.4.1 Correct LiDAR simulation

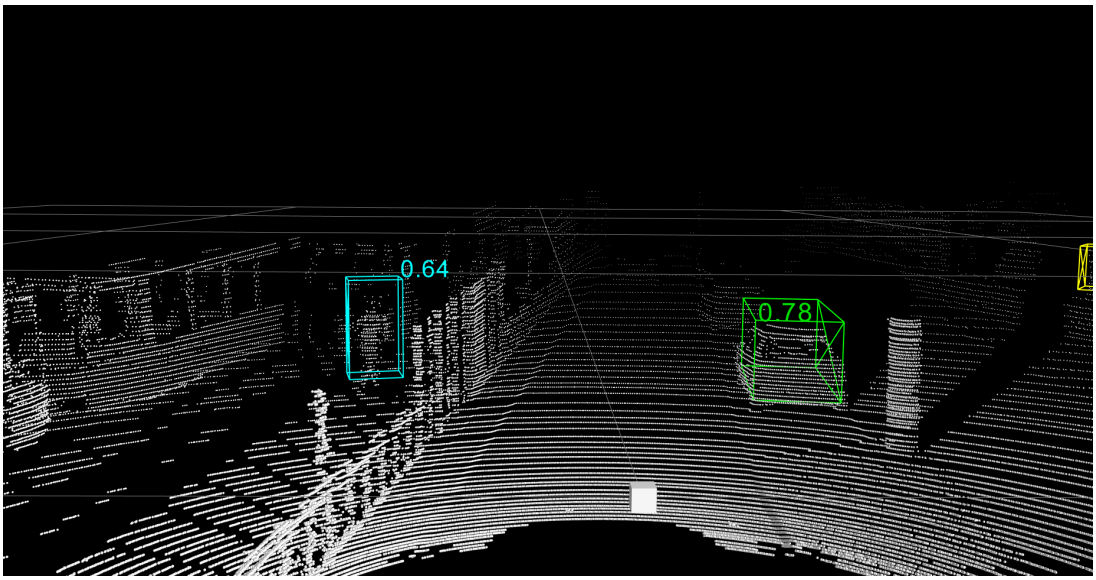
In this experiment we tried to show the importance of a correct simulation of the used LiDAR. Therefore we trained 4 different models which differ in amount of emitted beams. The model with 64 beams simulated the used LiDAR accurately, as the Velodyne HDL-64E emmits 64 beams. All models were trained with a drop-out of 30% and a noise of 1 cm, which we chose as standard levels. As can be seen in Table 2 the model which simulated the LiDAR correctly, i. e. the model with 64 beams, was the best performing one. Additionally the table shows, that the further the simulated LiDAR deviates from the correct simulation of the LiDAR the worse the performance was.

Model	**Car**
32 Beams, 30% Drop-Out, 1 cm Noise	11.85
48 Beams, 30% Drop-Out, 1 cm Noise	16.72
64 Beams, 30% Drop-Out, 1 cm Noise	**17.24**
96 Beams, 30% Drop-Out, 1 cm Noise	15.68
CDN [21]	19.0

Table 2: Performances of different models showing the impact of the correct simulation of the LiDAR. (64 Beams match target LiDAR)

The performance of the 64 Beam model was with 17.24% close to the 19.0% of the CDN [21] model, which we used as a reference. This shows, that our 3D object detector trained on the synthetic dataset, which we generated from the Apollo Synthetic dataset, is in line with other approaches utilizing synthetic data to train a 3D object detector. In general the poor performance of under 20% is referable to the inability of the object detector to accurately

detect objects if they are farther away. For nearby objects the accuracy of the object detector was better. Figure 15a visualizes this observation that the object detector accurately detected nearby cars but was unable to detect cars after a certain distance.



(a)

(b)

Figure 15: (a) Predicted bounding boxes by the on synthetic data trained 64 Beams model on a point cloud from the KITTI dataset. (b) Image from the KITTI dataset corresponding to the point cloud shown in (a).

4.4.2 Real World Effects

The two real world effects we focused on are noise and drop-out. Both effects are present in real world datasets, but are not present in synthetic data. This is due to the inability to simulate such real world effects in virtual environments. Therefore both effects contribute to the afforementioned domain gap and we tried to show the impact the simulation of these real world effects has on the performance of the 3D object detector.

Noise

Noise describes the effect of points deviating from their real position. This deviation can occur due to many real world properties, like for example the reflectance of the hit material or the lighting. As this reflectance of the material as well as the lighting is not accurately simulated in virual worlds noise is not present in the synthetic data. Therefore we tried to simulate noise by adding normally distributed noise to the points of the point cloud.

For the noise experiment we trained four different models with noise ranging from 0 cm to 2 cm. We chose the upper noise limit to be 2 cm, as the Velodyne HDL-64E LiDAR has an accuracy of smaller than 2 cm and we therefore assumed the noise with the best performance improvement to lie inbetween 0 and 2 cm. As can be seen in Table 3 this assumption held, with the model with 1 cm noise being the best performing one with an accuracy of 19.57%. This was an improvement of +2% in comparison to the 17.44% accuracy of the model without noise.

Model	**Car**
64 Beams, 0 cm Noise	17.44
64 Beams, 0.5 cm Noise	16.95
64 Beams, 1 cm Noise	**19.57**
64 Beams, 2 cm Noise	17.12
CDN [21]	19.0

Table 3: Performances of models showing the imact of simulated noise.

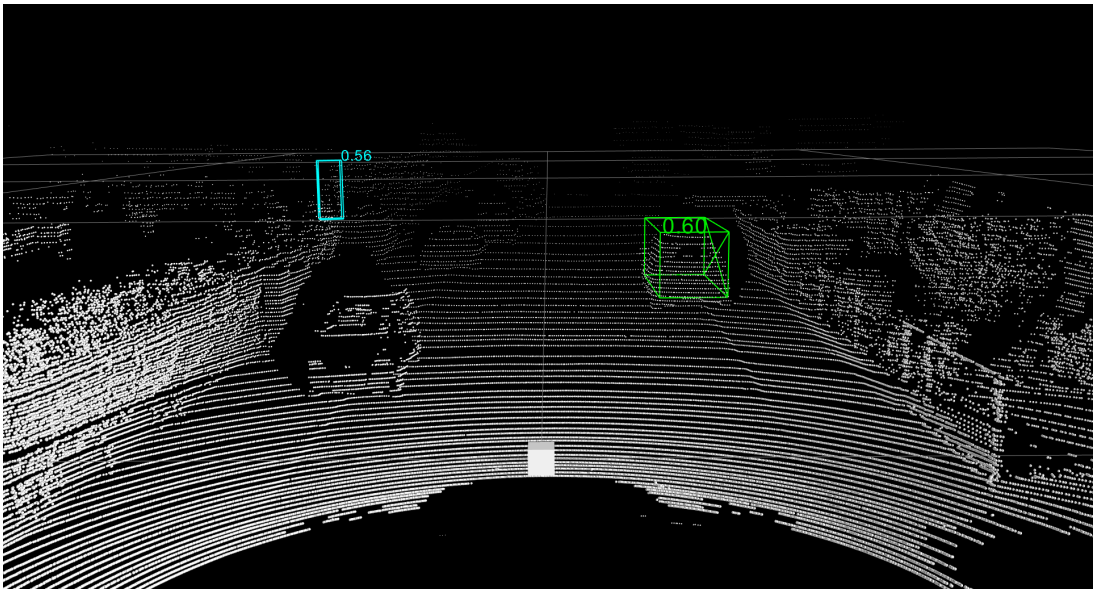
Drop-Out
Drop-Out describes the effect of missing points in the point cloud. Missing points can be caused by the texture of the hit material as well as its reflectance and absorptance. As mentioned before these material-specific effects can not be simulated accurately in the virtual worlds and are therefore missing in synthetic datasets. To simulate drop-out in our experiments we randomly discarded a certain percentage of points from the point cloud.

As the drop-out is not correlated to a LiDAR characteristic we could not assume a range as we did in the noise simulation experiment. Therefore we started with a drop-out of 30%, meaning we randomly discarded 30% of the points and increased it by 10% until we saw a drop-off in accuracy. Table 4 shows, that such a drop-off in accuracy can be seen from the 40% to the 50% drop-out model. Therefore we trained another model with 45% drop-out, which led to our best performing model with an accuracy of 23.45%. This was a performance improvement of roughly 6% compared to the model with no noise and no drop-out.

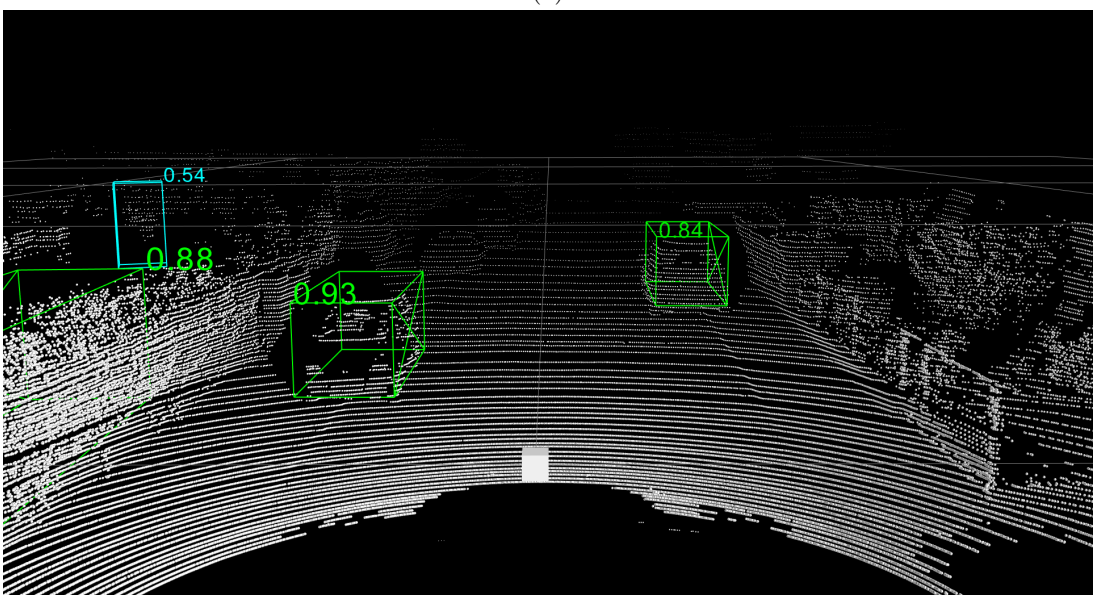
Model	**Car**
64 Beams, 30% Drop-Out	17.13
64 Beams, 40% Drop-Out	19.28
64 Beams, 45% Drop-Out	**23.45**
64 Beams, 50% Drop-Out	18.54
64 Beams, 0 cm Noise, no Drop-Out	17.44
64 Beams, 1 cm Noise, no Drop-Out	19.57
CDN [21]	19.0

Table 4: Performances of models showing the impact of simulated dropout.

This performance improvement due to the addition of drop-out can also be seen in the evaluation of a point cloud affected by missing points. Figure 16a shows such a point cloud, where the left front car is greatly affected by missing points in the front section of the car. Figure 16a visualizes the evaluation of this point cloud by the 64 Beams model with 1 cm noise but without drop-out. This model did not recognize the car due the high amount of missing points. When evaluating the same point cloud with the best drop-out model, i. e. the model trained with 45% drop-out, the model accurately predicted the bounding box for the car, as can be seen in Figure 16b.



(a)



(b)

(c)

Figure 16: (a) Predicted bounding boxes by the on synthetic data trained 64 Beams model with 1 cm noise and **without** drop-out on a point cloud from the KITTI dataset with a car (left front) affected by missing points. (b) Predicted bounding boxes by the on synthetic data trained 64 Beams model **with** 45% drop-out on a point cloud from the KITTI dataset with a car (left front) affected by missing points. (c) Image from the KITTI dataset corresponding to the point cloud shown in (a) and (b).

4.4.3 Anchor Sizes

In our last experiment we want to show the importance of using the dataset specific anchor sizes when evaluating a model. As aforementioned the anchor sizes are defined in the config-file of the model itself and describe the average size of a bounding box for the specific class. As was shown by Wang et al. [24] car sizes differ between Europe and America and therefore the car sizes differ between the Apollo Synthetic dataset, as it mainly consists of American car models, and the KITTI dataset, which was recorded in Germany. Therefore we need to use the anchor sizes of the KITTI dataset when evaluating our model to get the best results. Table 5 shows the differences in accuracy when evaluating the same model with different anchor sizes. As can be seen the evaluation with the correct anchor sizes, i. e. the anchor sizes of the KITTI dataset, reaches the highest accuracy. When evaluating the same model with slightly larger anchor sizes, i. e. the anchor sizes of the Apollo Synthetic dataset, the accuracy drops slightly. A bigger drop in accuracy can be seen, when using smaller anchor sizes for the evaluation. As the anchor sizes influence the sizes of the predicted bounding boxes the drop in performance can be explained with the evaluation metric. The used evaluation metric is 3D IoU, which measures the overlap of the predicted bounding box and the ground truth bounding box divided by their union. Therefore when a predicted bounding box deviates in size from the ground truth bounding box either the overlap decreases or the union increases. Both variants lead to a lower IoU and therefore a decrease in accuracy.

Model	**Car**
64 Beams, 1cm Noise, correct Anchor Sizes	**19.57**
64 Beams, 1cm Noise, bigger Anchors Sizes	17.46
64 Beams, 1cm Noise, smaller Anchors Sizes	13.79

Table 5: Performances of models showing the impact of using the right anchor sizes for the evaluation.

5 Conclusion & Future Work

Synthetically generated 3D object detection datasets are a promising mean to eliminate the need for real world 3D object detection datasets, which are expensive and labourious to create in the real world. Additionally synthetic data generation is adaptive to the characteristics of the capturing device, the LiDAR, as well as the environment, as both can be changed to fit the needs of the application. We were able to show the importance of the accurate LiDAR simulation for the synthetic data generation. The results show, that the accuracy of the 3D object detector drops, as the used LiDAR simulation deviates from its correct simulation. Further we show, that the simulation of real world effects like noise and drop-out can increase the accuracy of the 3D object detector, when evaluated on real world data.

Although our results are in line with state of the art approaches the accuracy of a 3D object detector trained on a synthetic dataset is still far away from the accuracy of a 3D object detector trained on a real world dataset. This gap in accuracy is caused by the domain gap between the synthetic and the real world data as it is not yet possible to realistically simulate the world in a virtual environment. Introducing real world effects like noise or drop-out can minimize this domain gap and therefore increase the accuracy of the object detector.

To further increase the accuracy of a synthetic dataset the environment would need to be modeled and simulated more realistically. This includes a more realistic simulation of materials including their refectance and absorptance values. Additionally the realistic simulation of light as well as the accurate simulation of small objects like grass or the surface of objects is important for generating realistic synthetic data.

Another possibility to improve the performance could be to utilize domain adaption approaches, such as the ones introduced previously, to adapt the object detector, trained on synthetic data, to the real world data. With this the domain gap could be minimized, which should lead to better performances.

References

[1] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on Robot Learning*, 2017. 4, 5

[2] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual worlds as proxy for multi-object tracking analysis. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016. 6

[3] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research*, 2013. 3, 4, 5, 20

[4] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012. 5, 11

[5] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014. 6

[6] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2 edition, 2004. 8, 12, 13

[7] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018. 4, 6, 8

[8] Braden Hurl, Krzysztof Czarnecki, and Steven Waslander. Precise synthetic image and lidar (presil) dataset for autonomous vehicle perception. In *IEEE Intelligent Vehicles Symposium*, 2019. 5

[9] Jacob Lambert, Alexander Carballo, Abraham Monrroy Cano, Patiphon Narksri, David Wong, Eijiro Takeuchi, and Kazuya Takeda. Performance analysis of 10 models of 3d lidars for automated driving. *IEEE Access*, 2020. 14

[10] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 4, 7, 19, 20

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, 2016. 19

[12] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael Jordan. Learning transferable features with deep adaptation networks. In *International Conference on Machine Learning*, 2015. 7

[13] Rafael Padilla, Sergio L Netto, and Eduardo AB da Silva. A survey on performance metrics for object-detection algorithms. In *International Conference on Systems, Signals and Image Processing*, 2020. 22

[14] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017. 7, 19

[15] Can Qin, Haoxuan You, Lichen Wang, C-C Jay Kuo, and Yun Fu. Pointdan: A multi-scale 3d domain adaption network for point cloud representation. *Conference on Neural Information Processing Systems*, 2019. 7, 23

[16] Stephan R Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision*, 2016. 3, 4, 5

[17] Khaled Saleh, Ahmed Abobakr, Mohammed Attia, Julie Iskander, Darius Nahavandi, Mohammed Hossny, and Saeid Nahvandi. Domain adaptation for vehicle detection from bird's eye view lidar point cloud data. In *IEEE International Conference on Computer Vision Workshops*, 2019. 7, 23

[18] Shaoshuai Shi, Chaoxu Guo, Li Jiang, Zhe Wang, Jianping Shi, Xiaogang Wang, and Hongsheng Li. Pv-rcnn: Point-voxel feature set abstraction for 3d object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 8

[19] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2019. 8

[20] Donghyeop Shin and Incheol Kim. Deep neural network-based scene graph generation for 3d simulated indoor environments. *KIPS Transactions on Software and Data Engineering*, 2019. 22

[21] Peng Su, Kun Wang, Xingyu Zeng, Shixiang Tang, Dapeng Chen, Di Qiu, and Xiaogang Wang. Adapting object detectors with conditional domain normalization. *European Conference on Computer Vision*, 2020. 3, 7, 23, 25

[22] OpenPCDet Development Team. Openpcdet: An open-source toolbox for 3d object detection from point clouds. *https://github.com/open-mmlab/OpenPCDet*, 2020. 19, 20

[23] Fei Wang, Yan Zhuang, Hong Gu, and Huosheng Hu. Automatic generation of synthetic lidar point clouds for 3-d data analysis. *IEEE Transactions on Instrumentation and Measurement*, 2019. 5

[24] Yan Wang, Xiangyu Chen, Yurong You, Li Erran Li, Bharath Hariharan, Mark Campbell, Kilian Q Weinberger, and Wei-Lun Chao. Train in germany, test in the usa: Making 3d object detectors generalize. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 3, 21, 27

[25] Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In *IEEE International Conference on Robotics and Automation*, 2018. 4, 5

[26] Yan Yan, Yuxing Mao, and Bo Li. Second: Sparsely embedded convolutional detection. *Sensors*, 2018. 7, 21

[27] Bin Yang, Wenjie Luo, and Raquel Urtasun. Pixor: Real-time 3d object detection from point clouds. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 2

[28] Zetong Yang, Yanan Sun, Shu Liu, and Jiaya Jia. 3dssd: Point-based 3d single stage object detector. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 8

[29] Faisal Zaman, Ya Ping Wong, and Boon Yian Ng. Density-based denoising of point cloud. In *9th International Conference on Robotic, Vision, Signal Processing and Power Applications*, 2017. 3

[30] Sicheng Zhao, Xiangyu Yue, Shanghang Zhang, Bo Li, Han Zhao, Bichen Wu, Ravi Krishna, Joseph E Gonzalez, Alberto L Sangiovanni-Vincentelli, Sanjit A Seshia, et al. A review of single-source deep unsupervised visual domain adaptation. *IEEE Transactions on Neural Networks and Learning Systems*, 2020. 6

[31] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 7, 8