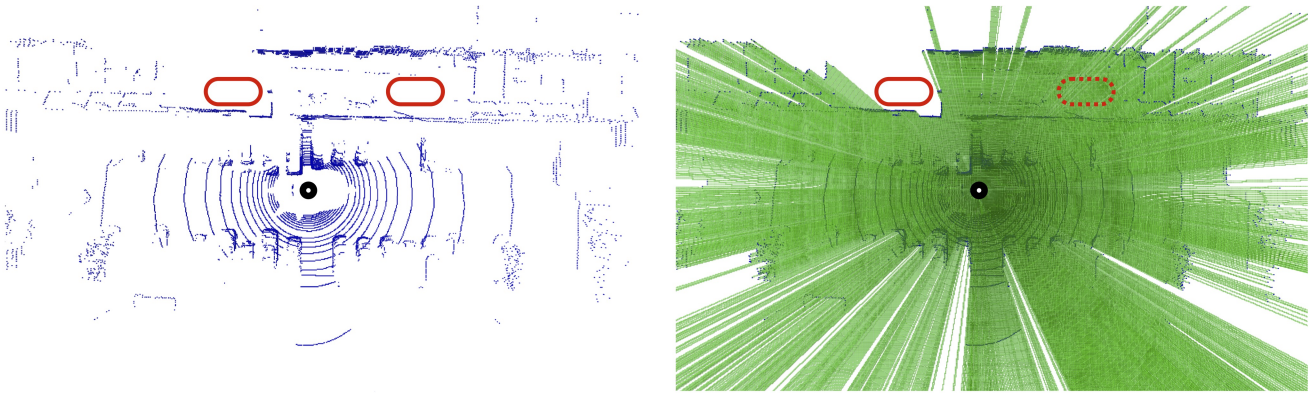
What You See is What You Get: Exploiting Visibility for 3D Object Detection

Peiyun Hu[1], Jason Ziglar[2], David Held[1], Deva Ramanan[1,2]
[1] Robotics Institute, Carnegie Mellon University
[2] Argo AI
peiyunh@cs.cmu.edu, jziglar@argo.ai, dheld@andrew.cmu.edu, deva@cs.cmu.edu



(a) (b)

Figure 1: What is a good representation for 3D sensor data? We visualize a birds-eye-view LiDAR scene and highlight two regions that may contain an object. Many contemporary deep networks process 3D point clouds, making it hard to distinguish the two regions (**left**). But depth sensors provide more than 3D points - they provide estimates of freespace in between the sensor and the measured 3D point. We visualize freespace by raycasting (**right**), where green is free and white is unknown. In this paper, we introduce deep 3D networks that leverage freespace to significantly improve 3D object detection accuracy.

Abstract

Recent advances in 3D sensing have created unique challenges for computer vision. One fundamental challenge is finding a good representation for 3D sensor data. Most popular representations (such as PointNet) are proposed in the context of processing truly 3D data (e.g. points sampled from mesh models), ignoring the fact that 3D sensored data such as a LiDAR sweep is in fact 2.5D. We argue that representing 2.5D data as collections of (x, y, z) points fundamentally destroys hidden information about freespace. In this paper, we demonstrate such knowledge can be efficiently recovered through 3D raycasting and readily incorporated into batch-based gradient learning. We describe a simple approach to augmenting voxel-based networks with visibility: we add a voxelized visibility map as an additional input stream. In addition, we show that visibility can be combined with two crucial modifications common to state-of-the-art 3D detectors: synthetic data augmentation of virtual objects and temporal aggregation of LiDAR sweeps over multiple time frames. On the NuScenes 3D detection benchmark, we show that, by adding an additional stream for visibility input, we can significantly improve the overall detection accuracy of a state-of-the-art 3D detector.

1. Introduction

What is a good representation for processing 3D sensor data? While this is a fundamental challenge in machine vision dating back to stereoscopic processing, it has recently been explored in the context of deep neural processing of 3D sensors such as LiDARs. Various representations have been proposed, including graphical meshes [2], point clouds [21], voxel grids [34], and range images [19], to name a few.

Visibility: We revisit this question by pointing out that 3D sensored data, is infact, not fully 3D! Instantaneous depth measurements captured from a stereo pair, structured light sensor, or LiDAR undeniably suffer from occlusions:

once a particular scene element is measured at a particular depth, visibility ensures that all other scene elements behind it along its line-of-sight are occluded. Indeed, this loss of information is one of the fundamental reasons why 3D sensor readings can often be represented with 2D data structures - e.g., 2D range image. From this perspective, such 3D sensored data might be better characterized as "2.5D" [18].

3D Representations: We argue that representations for processing LiDAR data should embrace visibility, particularly for applications that require instantaneous understanding of freespace (such as autonomous navigation). However, most popular representations are based on 3D point clouds (such as PointNet [21, 14]). Because these were often proposed in the context of truly 3D processing (e.g., of 3D mesh models), they do not exploit visibility constraints implicit in the sensored data (Fig. 1). Indeed, representing a LiDAR sweep as a collection of (x, y, z) points fundamentally *destroys* such visibility information if normalized (e.g., when centering point clouds).

Occupancy: By no means are we the first to point out the importance of visibility. In the context of LiDAR processing, visibility is well studied for the tasks of map-building and occupancy reasoning [27, 8]. However, it is not well-explored for object detection, with one notable exception: [33] builds a probabilistic occupancy grid and performs template matching to directly estimate the probability of an object appearing at each discretized location. However, this approach requires knowing surface shape of object instances beforehand, therefore it is not scalable. In this paper, we demonstrate that deep architectures can be simply augmented to exploit visibility and freespace cues.

Range images: Given our arguments above, one solution might be defining a deep network on 2D range image input, which *implicitly* encodes such visibility information. Indeed, this representation is popular for structured light "RGBD" processing [10, 6], and has also been proposed for LiDAR [19]. However, such representations do not seem to produce state-of-the-art accuracy for 3D object understanding, compared to 3D voxel-based or top-down, birds-eye-view (BEV) projected grids. We posit that convolutional layers that operate along a depth dimension can reason about uncertainty in depth. To maintain this property, we introduce simple but novel approaches that directly augment state-of-the-art 3D voxel representations with visibility cues.

Our approach: We propose a deep learning approach that efficiently augments point clouds with visibility. Our specific constributions are three-fold; (1) We first (re)introduce raycasting algorithms that effciently compute on-the-fly visibility for a voxel grid. We demonstrate that these can be incorporated into batch-based gradient learning. (2) Next, we describe a simple approach to augmenting voxel-based networks with visibility: we add a voxelized visibility map as an additional input stream, exploring alternatives for early and late fusion; (3) Finally, we show that visibility can be combined with two crucial modifications common to state-of-the-art networks: synthetic data augmentation of virtual objects, and temporal aggregation of LiDAR sweeps over multiple time frames. We show that visibility cues can be used to better place virtual objects. We also demonstrate that visibility reasoning over multiple time frames is akin to online occupancy mapping.

2. Related Work

2.1. 3D Representations

Point representation Most classic works on point representation employ *hand-crafted* descriptors and require robust estimates of local surface normals, such as spin-images [9] and Viewpoint Feature Histograms (VFH) [23]. Since PointNet [21], there has been a line of work focuses on learning better point representation, including PointNet++[22], Kd-networks [12], PointCNN [15], EdgeConv [29], and PointConv [30] to name a few. Recent works on point-wise representation tend not to distinguish between *reconstructed* and *measured* point clouds. We argue that when the input is a *measured* point cloud, e.g. a LiDAR sweep, we need to look beyond points and reason about visibility that is hidden within points.

Visibility representation Most research on visibility representation has been done in the context of robotic mapping. For example, Buhmann et al. [3] estimates a 2D probabilistic occupancy map from sonar readings to navigate the mobile robot and more recently Hornung et al. [8] have developed Octomap for general purpose 3D occupancy mapping. Visibility through raycasting is at the heart of developing such occupancy maps. Despite the popularity, such visibility reasoning has not been widely studied in the context of object detection, except a notable exception of [33], which develops a probabilistic framework based on occupancy maps to detect objects with known surface models.

2.2. LiDAR-based 3D Object Detection

Initial representation We have seen LiDAR-based object detectors built upon range images, bird-eye-view feature maps, raw point clouds, and also voxelized point clouds. One example of a range image based detector is LaserNet [19], which treats each LiDAR sweep as a cylindrical range image. Examples of bird-eye-view detectors include AVOD [13], HDNet [32], and Complex-YOLO [25]. One example that builds upon raw point clouds is PointRCNN [24]. Examples of voxelized point clouds include the initial VoxelNet[34], SECOND [31], and PointPillars [14]. Other than [33], we have not seen a detector that uses visibility as the initial representation.

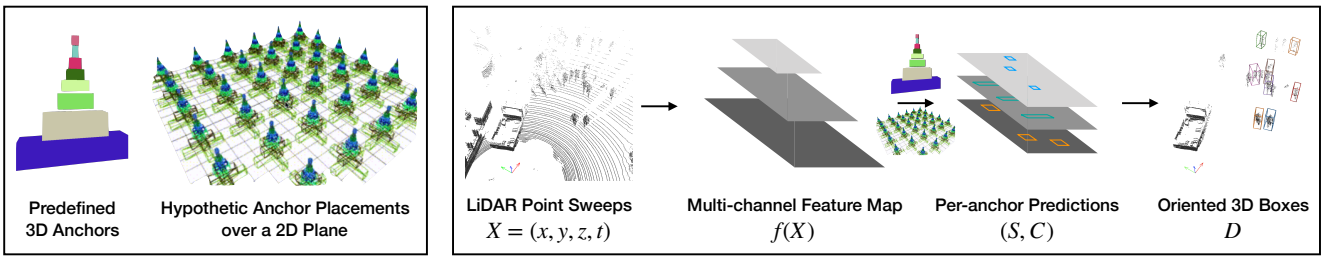


Figure 2: Overview of a general 3D detection framework, designed to solve 3D detection as a bird-eye-view (BEV) 2D detection problem. The framework consists of two parts: anchors (**left**) and network (**right**). We first define a set of 3D anchor boxes that match the average box shape of different object classes. Then we hypothesize placing each anchor at different spatial locations over a ground plane. We learn a convolutional network to predict confidence and adjustments for each anchor placement. Such predictions are made based on 2D multi-channel feature maps, extracted from the input 3D point cloud. The predictions for each anchor consist of a confidence score S and a set of coefficients C for adjusting the anchor box. Eventually, the framework produces a set of 3D detections with oriented 3D boxes.

Object augmentation Yan et al. [31] propose a novel form of data augmentation, which we call *object augmentation*. It copy-pastes object point clouds from one scene into another, resulting in new training data. This augmentation technique improves both convergence speed and final performance and is adopted in all recent state-of-the-art 3D detectors, such as PointRCNN [24], PointPillars [14]. For objects captured under the same sensor setup, simple copy-paste preserves the relative pose between the sensor and the object, resulting in approximately correct return patterns. However, such practice often inserts objects regardless of whether it violates the scene visibility. In this paper, we propose to use visibility reasoning to maintain correct visibility while augmenting objects across scenes.

Temporal aggregation When learning 3D object detectors over a series of LiDAR sweeps, it is proven helpful to aggregate information across time. Luo et al. [17] develops a recurrent architecture for detecting, tracking, and forecasting objects on LiDAR sweeps. Choy et al. [5] proposes to learn spatial-temporal reasoning through 4D ConvNets. Another technique for temporal aggregation, first found in SECOND [31], is to simply aggregate point clouds from different sweeps while preserving their timestamps relative to the current one. These timestamps are treated as additional per-point input feature along with (x, y, z) and fed into point-wise encoders such as PointNet. We explore temporal aggregation over visibility representations and point out that one can borrow ideas from classic robotic mapping to integrate visibility representation with learning.

3. Exploit Visibility for 3D Object Detection

Before we discuss how to integrate visibility reasoning into 3D detection, we first introduce a general framework for 3D detection. Many 3D detectors have adopted this framework, including AVOD [13], HDNet [32], Complex-YOLO [25], VoxelNet [34], SECOND [31], and PointPillars [14]. Among the more recent ones, there are two crucial innovations: (1) object augmentation by inserting rarely seen (virtual) objects into training data and (2) temporal aggregation of LiDAR sweeps over multiple time frames.

We integrate visibility into the aforementioned 3D detection framework. First, we (re)introduce a raycasting algorithm that efficiently computes visibility. Then, we introduce a simple approach to integrate visibility into the existing framework. Finally, we discuss visibility reasoning within the context of object augmentation and temporal aggregation. For object augmentation, we modify the raycasting algorithm to make sure visibility remains intact while inserting virtual objects. For temporal aggregation, we point out that visibility reasoning over multiple frames is akin to online occupancy mapping.

3.1. A General Framework for 3D Detection

Overview We visualize a general framework for 3D detection in Fig. 2. Please refer to the caption. We highlight the fact that once the input 3D point cloud is converted to a multi-channel BEV 2D representation, we can make use of standard 2D convolutional architectures. We later show that visibility can be naturally incorporated into this 3D detection framework.

Object augmentation Data augmentation is a crucial ingredient of contemporary training protocols. Most augmentation strategies perturb coordinates through random transformations (e.g. translation, rotation, flipping) [13, 20]. We focus on *object augmentation* proposed by Yan et al. [31], which copy-pastes (virtual) objects of rarely-seen classes (such as buses) into LiDAR scenes. Our ablation studies (g→i in Tab. 3) suggest that it dramatically improves

vanilla PointPillars by an average of **+9.1%** on the augmented classes.

Temporal aggregation In LiDAR-based 3D detection, researchers have explored various strategies for temporal reasoning. We adopt a simple method that aggregates (motion-compensated) points from different sweeps into a single scene [31, 4]. Importantly, points are augmented with an additional channel that encodes its relative timestamp (x, y, z, t). Our ablation studies (g→j in Tab. 3) suggest that temporal aggregation dramatically improves the overall mAP of vanilla PointPillars model by **+8.6%**.

3.2. Compute Visibility through Raycasting

Physical raycasting in LiDAR Each LiDAR point is generated through a physical raycasting process. To generate a point, the sensor emits a laser pulse in a certain direction. The pulse travels forward through air and back after hitting an obstacle. Upon its return, one can compute a 3D coordinate derived from the the direction and time-of-flight. However, coordinates are by no means the only information offered by such sensing. Crucially, active sensing also provides estimates of freespace along ray traveled by the pulse.

Simulated LiDAR raycasting By exploiting the causal relationship between freespace and point returns - points lie along the ray where freespace ends, we can re-create the instantaneous visibility encountered at the time of LiDAR capture. We do so by drawing a line segment from the sensor origin to every 3D point. We would like to use this line segment to define freespace across a discretized volume, e.g. a 3D voxel grid. Specifically, we compute all voxels that intersect this line segment. Those that are encountered along the way are marked as free, while the last voxel enclosing the 3D point is occupied. This results in a visibility volume where all voxels are marked as occupied, free, or unknown (default). We will integrate the visiblity volume into the general detection framework (Fig. 2) in the form of a *multi-channel 2D feature map* where visibility along the vertical dimension (z-axis) is treated as multiple channels.

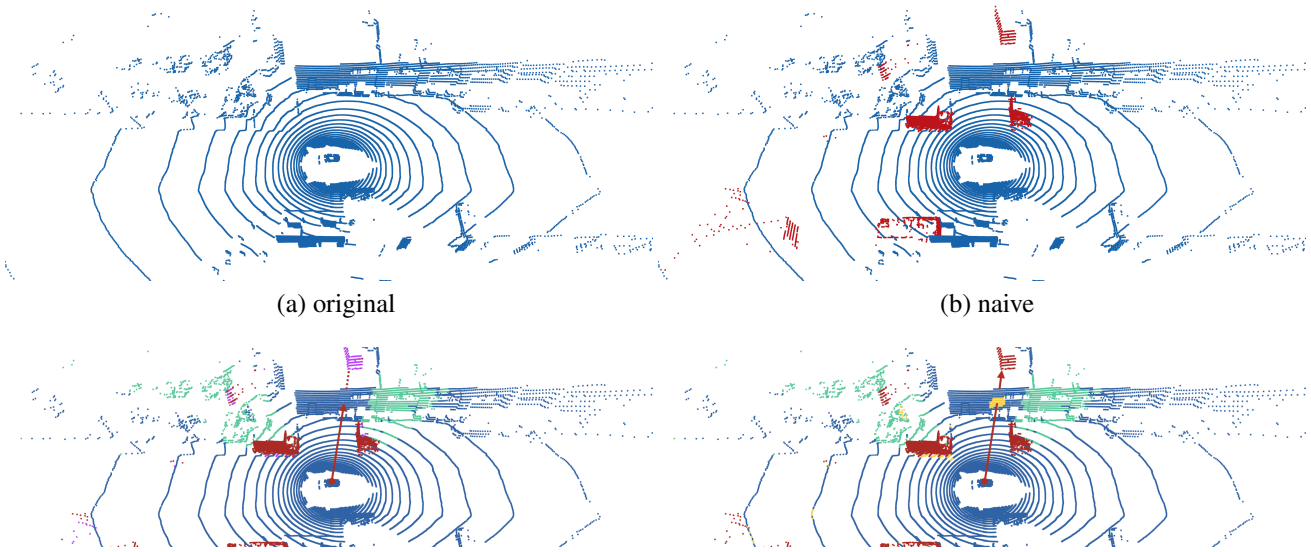
Efficient voxel traversal In order to ensure fast inference times and efficient training times, our visibility computation must be extremely efficient. Many detection networks exploit sparsity in LiDAR point clouds: PointPillars[14] process only non-empty pillars (about 3%) and SECOND [31] employs spatially sparse 3D ConvNets. Inspired by these approaches, we exploit sparsity through an efficient voxel traversal algorithm [1]. For any given ray, we need traverse only those sparse set of voxels that intersect with the ray. Intuitively, during the traversal, the algorithm enumerates over the six axis-aligned faces of the current voxel to determine which is intersected by the exiting ray (which is quite efficient). It then simply advances to the neighboring voxel with a shared face. The algorithm begins at the voxel at the origin and terminates when it encounters the (precomputed) voxel occupied by the 3D point. This algorithm is linear in the resolution of a single grid dimension, making it quite efficient. We perform raycasting of multiple points in parallel and aggregate computed visibility afterwards. We also follow best-practices outlined in Octomap (Sec. 5.1 in [8]) to reduce discretization effects during aggregation.

Raycasting with augmented objects Prior work augments virtual objects while ignoring visibility constraints, producing inconsistent LiDAR sweeps (e.g., by inserting an object behind a wall that should occlude it - Fig. 3-(b)). We can use ray-casting as a tool to "rectify" the LiDAR sweep. Specifically, we might wish to remove virtual objects that are occluded (a strategy we term *culling* - Fig. 3-(c)). Because this might excessively decrease the number of augmented objects, another option is to remove points from the original scene that occlude the inserted objects (a strategy we term *drilling* - Fig. 3-(d)). Fortunately, both strategies are efficient to implement with simple modifications to the above ray-casting algorithm. We only have to change the terminating condition of raycasting from arriving at the end point of the ray to hitting a voxel that is *pre-occupied*. When casting rays from the original scene, we set voxels occupied by virtual objects as *pre-occupied*. And when casting rays from the virtual objects, we set voxels occupied by original scenes as *pre-occupied*. As a consequence, points that should be occluded will be removed.

Online occupancy mapping How do we extend instantaneous visibility into a temporal context? Assume knowing the sensor origin at each timestamp, we can compute instantaneous visibility over every sweep, resulting in 4D spatial-temporal visibility. If we directly integrate a 4D volume into the detection framework, it would be too expensive. We seek out online occupancy mapping [28, 8] and apply Bayesian filtering to turn a 4D spatial-temporal visibility into a 3D posterior probability of occupancy. In Fig. 4, we plot a visual comparison between instantaneous visibility and temporal occupancy. We follow Octomap [8]'s formulation and use their off-the-shelf hyper-parameters, e.g. the log-odds of observing freespace and occupied space.

3.3. Approach: A Two-stream Network

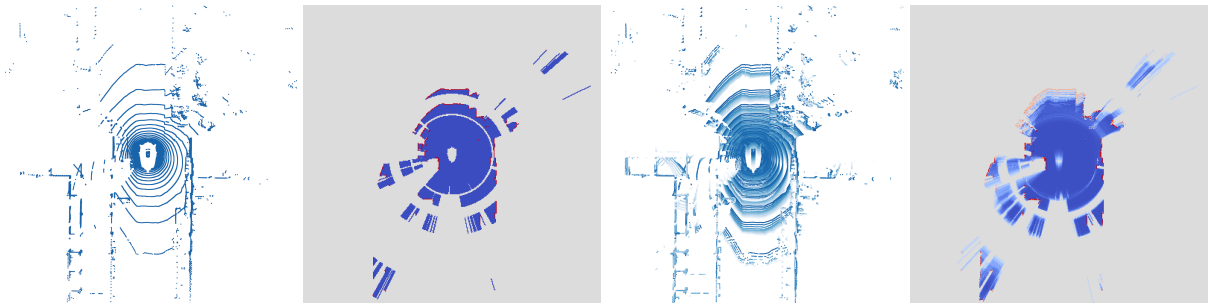
Now that we have discussed raycasting approaches for computing visibility, we introduce a novel two-stream network for 3D object detection. We add an additional stream for visibility input into the network of a state-of-the-art 3D detector, i.e. PointPillars. As a result, our approach leverages both point cloud representation and visibility representation and fuses them into a multi-channel representation.



(a) original (b) naive

(c) culling (d) drilling

Figure 3: Different types of object augmentation we can do through visibility reasoning. In (a), we show the original LiDAR point cloud. In (b), we naively insert new objects (red) into the scene. Clearly, the naive strategy may result in inconsistent visibility. Here, a trailer is inserted behind a wall that should occlude it. We use raycasting as a tool to "rectify" the LiDAR sweep. In (c), we illustrate the culling strategy, where we remove virtual objects that are occluded (purple). In practice, this may excessively remove augmented objects. In (d), we visualize the drilling strategy, where we remove points from the original scene that occlude the virtual objects. Here, a small piece of wall is removed (yellow).

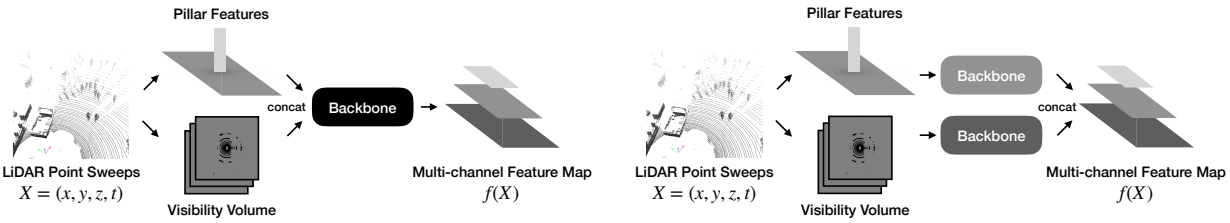


(a) instantaneous visibility (b) temporal occupancy

Figure 4: We visualize instantaneous visibility vs. temporal occupancy. We choose one xy-slice in the middle to visualize. Each pixel represents a voxel on the slice. On the **left**, we visualize a single LiDAR sweep and the instantaneous visibility, which consists of three discrete values: occupied (red), unknown (gray), and free (blue). On the **right**, we visualize aggregated LiDAR sweeps plus temporal occupancy, computed through Bayesian Filtering [8]. Here, the color encodes the probability of the corresponding voxel being occupied: redder means more occupied.

We explore two fusion strategies: early fusion and late fusion, as illustrated in Fig. 5. The overall network architecture follows the illustration in Fig. 2.

Implementation We implement our two-stream network by adding an additional input stream to PointPillars. We adopt PointPillar's resolution for discretization in order to improve ease of integration. As a consequence, our visibility volume has the same 2D spatial size as the pillar feature maps. A simple strategy is to concatenate these two first and then feed them into a backbone network. We refer to this strategy as early fusion (Fig. 5-(a)). Another strategy is to have a separate backbone network for both pillar feature maps and visibility volume, which we refer to as late fusion (Fig. 5-(b)). Please refer to the supplementary materials for more implementation details.



(a) early fusion (b) late fusion

Figure 5: We explore both early fusion and late fusion when integrating visibility into pointpillar models. In the early fusion (a), we concatenate visibility volume with pillar features before applying a backbone network for further encoding. For late fusion, we build a separate backbone network for each stream and concatenate the final multi-channel feature maps. We compare these two alternatives in ablation studies (Tab. 3).

Table 1: 3D detection mAP on the NuScenes test set.

	car	pedes.	barri.	traff.	truck	bus	trail.	const.	motor.	bicyc.	mAP
PointPillars [4]	68.4	59.7	**38.9**	**30.8**	23.0	28.2	23.4	4.1	**27.4**	**1.1**	30.5
Ours	**79.1**	**65.0**	34.7	28.8	**30.4**	**46.6**	**40.1**	**7.1**	18.2	0.1	**35.0**

4. Experiments

We present both qualitative (Fig. 6) and quantitative results on the NuScenes 3D detection benchmark. We first introduce the setup and baselines, before presenting the main results on the test benchmark. Afterwards, we perform diagnostic evaluation and ablation studies to isolate where improvements come from. Finally, we discuss the efficiency of computing visibility through raycasting on-the-fly.

Setup We benchmark our approach on the NuScenes 3D detection dataset. The dataset contains 1,000 scenes captured in two cities. We follow the official protocol for NuScenes detection benchmark. The training set contains 700 scenes (28,130 annotated frames). The validation set contains 150 scenes (6,019 annotated frames). Each annotated frame comes with one LiDAR point cloud captured by a 32-beam LiDAR, as well as up to 10 frames of (motion-compensated) point cloud. We follow the official evaluation protocol for 3D detection [4] and evaluate average mAP over different classes and distance threshold.

Baseline PointPillars [14] achieves the best accuracy on the NuScenes detection leaderboard among all published methods that have released source code. The official PointPillars codebase[1] only implements 3D detection on KITTI [7]. To reproduce PointPillars results on NuScenes, the authors of PointPillars recommend a third-party implementation[2].Using an off-the-shelf configuration provided by the third-part implementation, we train a PointPillars model for 20 epochs from scratch on the full training set and use it as our baseline. This model achieves an overall mAP of 31.5% on the validation set, which is 2% higher than the official PointPillars mAP (29.5%) [4] (Tab. 2). As suggested by [4], the official implementation of PointPillars employ pretraining (ImageNet/KITTI). There is no pretraining in our re-implementation.

Main results We submitted the results of our two-stream approach to the NuScenes test server. In Tab. 1, we compare our test-set performance to PointPillars on the official leaderboard [4]. By augmenting visibility, our proposed approach achieves a significant improvement over PointPillars in overall mAP by a margin of 4.5%. Specifically, our approach outperforms PointPillars by 10.7% on cars, 5.3% on pedestrians, 7.4% on trucks, 18.4% on buses, 16.7% on trailers. Our model underperforms official PointPillars on motorcycles by a large margin. We hypothesize this might be due to (1) a different configuration for PointPillars (e.g. xy-resolution) or (2) pretraining on ImageNet/KITTI.

Improvement at different levels of visibility We compare our two-stream approach to PointPillars on the validation set, where we see a 4% improvement from having visibility. We also evaluate each object class at different levels of visibility. Here, we plot results over the two most common classes: car and pedestrian. Interestingly, the biggest improvement are observed for cars that are heavily-occluded (0-40% visible) and the smallest improvements correspond to fully-visible cars (80-100% visible). For pedestrian, we also see the smallest improvement

[1] https://github.com/nutonomy/second.pytorch
[2] https://github.com/traveller59/second.pytorch

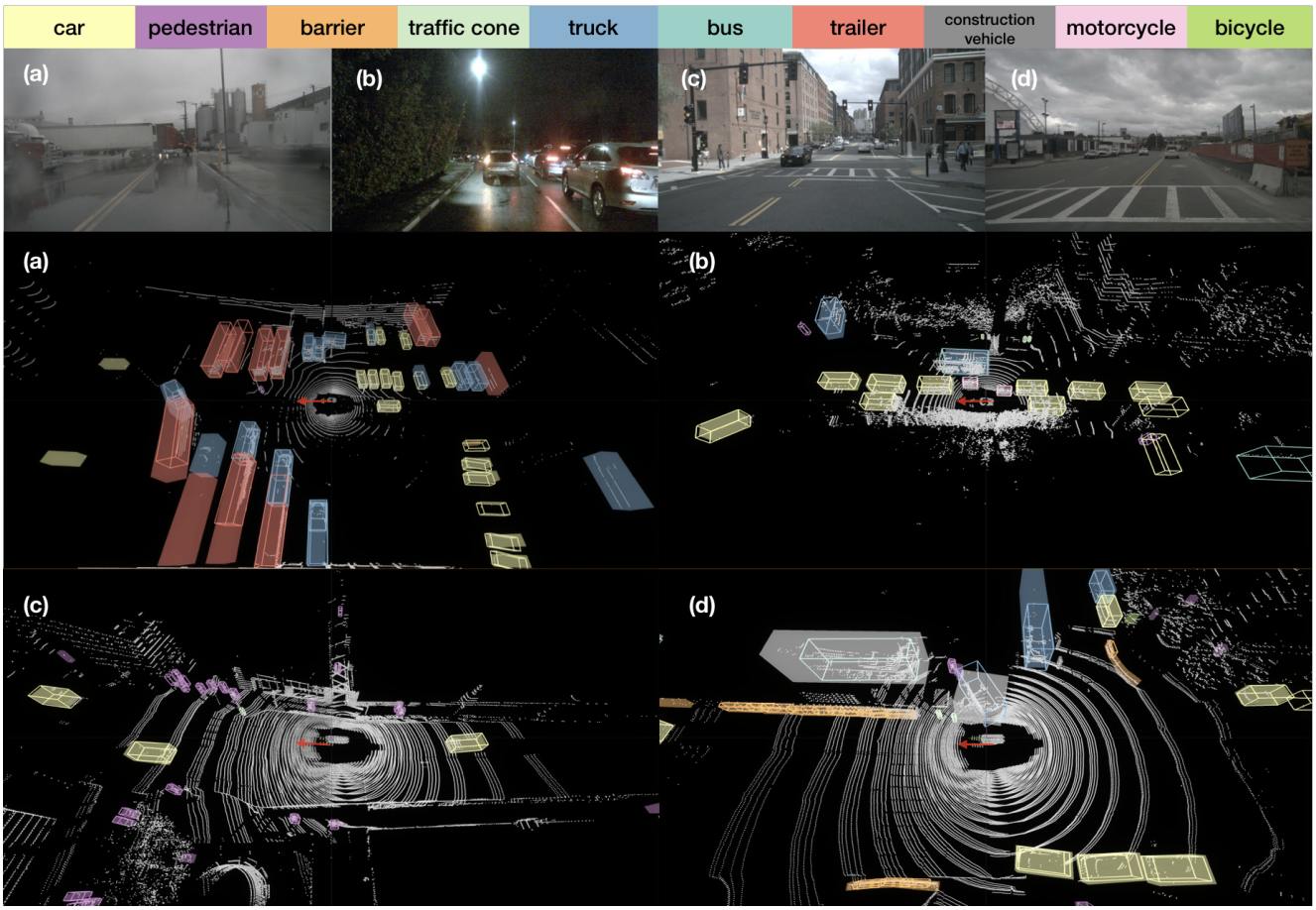


Figure 6: We visualize qualitative results from our two-stream approach on the test set of the NuScenes 3D detection benchmark. We visualize each class with a different color. We use solid transparent cuboids to represent ground truth objects and wireframe boxes to represent predictions. In all examples, the autonomous vehicle is facing the left. To provide context, we also include an image captured by the front camera for each scenario. Note the image is **not** used as part of input for our approach, to provide more context. In (a), our approach successfully detects most vehicles in the scene on a rainy day, including cars, trucks, and trailers. In (b), our model manages to detect all the cars around and also two motorcycles on the right side. In (c), we visualize a scene with many pedestrians on the sidewalk and our model is able to detect most of them. Finally, we demonstrate a failure case in (d), where our model fails to detect objects from rare classes. In this scenario, our model fails to detect two construction vehicles on the car's right side, reporting one as a truck and the other as a bus.

on fully-visible pedestrians (3.2%), which is 1-3% less than what we observe for pedestrians with more occlusion.

Ablation studies To understand how much improvement each component provides, we perform additional ablation studies. We start from our final model and remove one component at a time. Key observations from Tab. 3 are:

- (**a**, b) Replacing early fusion (a) with late fusion (b) results in a 1.4% drop in overall mAP.
- (**b**, c, d) Replacing drilling (b) with culling (c) results in a 11.4% drop on bus and a 4.9% drop on trailer. In practice, most augmented trucks and trailers tend to be severely occluded and are removed if the culling strategy is applied. Replacing drilling (b) with naive augmentation (d) results in a 1.9% drop on bus and 3.1% drop on trailer, likely due to inconsistent visibility when naively augmenting objects.
- (**b**, e) Removing object augmentation (b→e) leads to significant drops in mAP on classes affected by object augmentation, including in a 2.5% drop on truck, 13.7% on bus, and 7.9% on trailer.
- (**e**, f) Removing temporal aggregation (e→f) results in worse performance on every class and a 9.4% drop in overall mAP.

Table 2: 3D detection mAP on the NuScenes validation set.
†: reproduced based on an author-recommended third-party implementation.

	car	pedes.	barri.	traff.	truck	bus	trail.	const.	motor.	bicyc.	mAP
PointPillars [4]	70.5	59.9	33.2	**29.6**	25.0	34.4	16.7	4.5	**20.0**	**1.6**	29.5
PointPillars†	76.9	62.6	29.2	20.4	32.6	49.6	27.9	3.8	11.7	0.0	31.5
Ours	**80.0**	**66.9**	**34.5**	27.9	**35.8**	**54.1**	**28.5**	**7.5**	18.5	0.0	**35.4**

car	0-40%	40-60%	60-80%	80-100%
Proportion	20%	12%	15%	54%
PointPillars†	27.2	40.0	57.2	84.3
Ours	32.1	42.6	60.6	86.3
Improvement	**4.9**	2.6	3.4	2.0

pedestrian	0-40%	40-60%	60-80%	80-100%
Proportion	20%	12%	15%	54%
PointPillars†	17.3	23.4	28.0	68.3
Ours	22.1	27.8	34.2	71.5
Improvement	4.8	4.4	**6.2**	3.2

Table 3: Ablation studies on the NuScenes validation set. OA stands for object augmentation and TA stands for temporal aggregation. Classes affected by OA are *italicized*.

	Fusion	OA	TA	car	pedes.	barri.	traff.	*truck*	*bus*	*trail.*	const.	motor.	bicyc.	avg
(a)	Early	Drilling	Multi-frame	80.0	66.9	34.5	27.9	35.8	54.1	28.5	7.5	18.5	0.0	35.4
(b)	Late	Drilling	Multi-frame	77.8	65.8	32.2	24.2	33.7	53.0	30.6	4.1	18.8	0.0	34.0
(c)	Late	Culling	Multi-frame	78.3	66.4	33.2	27.3	33.4	41.6	25.7	5.6	17.0	0.1	32.9
(d)	Late	Naive	Multi-frame	78.2	66.0	32.7	25.6	33.6	51.1	27.5	4.7	15.0	0.1	33.5
(e)	Late	N/A	Multi-frame	77.9	66.8	31.3	22.3	31.2	39.3	22.7	5.2	15.5	0.6	31.3
(f)	Late	N/A	Single-frame	67.9	45.7	24.0	12.4	22.6	29.9	8.5	1.3	7.1	0.0	21.9
(g)	No V	N/A	Single-frame	68.0	38.2	20.7	8.7	23.7	28.7	11.0	0.6	5.6	0.0	20.5
(h)	Only V	N/A	Single-frame	66.7	28.6	15.8	4.4	17.0	25.4	6.7	0.0	1.3	0.0	16.6
(i)	No V	Naive	Single-frame	69.7	38.7	22.5	11.5	28.1	40.7	21.8	1.9	4.7	0.0	24.0
(j)	No V	N/A	Multi-frame	77.7	61.6	26.4	17.2	31.2	38.5	24.2	3.1	11.5	0.0	29.1
(k)	No V	Naive	Multi-frame	76.9	62.6	29.2	20.4	32.6	49.6	27.9	3.8	11.7	0.0	31.5

- (**f**, g, h) Removing visibility stream off a *vanilla* two-stream approach (f→g) drops overall mAP by 1.4%. Interestingly, the most dramatic drops are over pedestrian (+7.5%), barrier(+3.3%), and traffic cone (+3.7%). Shape-wise, these objects are all "skinny" and tend to have less LiDAR points on them. This suggests visibility helps especially when having less points. A single-stream network with only visibility (h) underperforms a *vanilla* PointPillars (g) by 4%.

- (**g**, i, j, k) Object augmentation (g→i) improves vanilla PointPillars (g) by an average of 9.1% in AP on augmented classes. Temporal aggregation (g→j) improves vanilla PointPillars by 8.6% in overall mAP. Adding both (g→k) increases the overall mAP by 11.0%.

Run-time We implement raycasting in C++ and call it from Python with the help of PyBind11. We integrate visibility computation into our PyTorch pipeline as part of dataloader pre-processing. With our implementation, it takes 24.4 ± 3.5 milliseconds on average to compute visibility over a 32-beam LiDAR point on an Intel i9-9980XE CPU.

Conclusions We revisit the problem of finding a good representation for 3D data. We point out that contemporary representations are designed for true 3D data (e.g. sampled from mesh models). In fact, 3D sensored data such as a LiDAR sweep is 2.5D. By processing such data as a collection of *normalized* points (x, y, z), important visibility information is fundementally destroyed. In this paper, we augment visibility into 3D object detection. We first demonstrate that visibility can be efficiently re-created through 3D raycasting. We introduce a simple two-stream approach that adds visibility as a separate stream to an existing state-of-the-art 3D detector. We also discuss the role of visibility in placing virtual objects for data augmentation and explore visibility in a temporal context - building a local occupancy map in an online fashion. Finally, on the NuScenes detection benchmark, we demonstrate that the proposed network outperforms state-of-the-art detectors by a significant margin.

A. Additional Qualitative Results

Please find a video (compressed or uncompressed) for additional qualitative results. In Fig. A, we use an example frame from the video to illustrate how we visualize.

B. Additional Method Details

Here, we provide additional details about our method, including pre-processing, network structure, initialization, loss function, training etc. These details apply to both the baseline method (PointPillars) and our two-stream approach.

pre-processing We focus on points whose (x, y, z) satisfies $x \in [-50, 50], y \in [-50, 50], z \in [-5, 3]$ and ignore points outside the range when computing pillar features. We group points into vertical columns of size $0.25 \times 0.25 \times 8$. We call each vertical column a pillar. We resample to make sure each non-empty pillar contains 60 points. For raycasting, we do **not** ignore points outside the range and use a voxel size of $0.25 \times 0.25 \times 0.25$.

Network structure We introduce (1) pillar feature network; (2) backbone network; (3) detection heads.

(1) Pillar feature network operates over each non-empty pillar. It takes points (x, y, z, t) within the pillar and produces a 64-d feature vector. To do so, it first compresses (x, y) to (r), where $r = \sqrt{x^2 + y^2}$. Then it augments each point with its offset to the pillar's arithmetic mean (x_c, y_c, z_c) and geometric mean (x_p, y_p). Please refer to Sec. 2.1 of PointPillars [14] for more details. Then, it processes augmented points $(r, z, t, x - x_c, y - y_c, z - z_c, x - x_p, y - y_p)$ with a 64-d linear layer, followed by BatchNorm, ReLU, and MaxPool, which results in a 64-d embedding for each non-empty pillar. Conceptually, this is equivalent to a mini one-layer PointNet. Finally, we fill empty pillars with all zeros. Based on our discretization choices, pillar feature network produces a $400 \times 400 \times 64$ feature map.

(2) Backbone network is a convolutional network with an encoder-decoder structure. This is also sometimes referred to as Region Proposal Network. Please read VoxelNet [34], SECOND [31], and PointPillars [14] for more details. The network consists three blocks of fully convolutional layers. Each block consists of a convolutional stage and a deconvolutional stage. The first (de)convolution filter of the (de)convolutional stage changes the spatial resolution and the feature dimension. All (de)convolution is 3x3 and followed with BatchNorm and ReLU. For our two-stream early-fusion model, the backbone network takes an input of size $400 \times 400 \times 96$, where 64 channels are from pillar feature and 32 channels are from visibility. The first block contains 4 convolutional layers and 4 deconvolutional layers. The second and the third block each consists of 6 both of these layers. Within the first block, the feature dimension changes from $400 \times 400 \times 96$ to $200 \times 200 \times 96$ during the convolutional stage, and $200 \times 200 \times 96$ to $100 \times 100 \times 192$ during the deconvolutional stage. Within the second block, the feature dimension from $200 \times 200 \times 96$ to $100 \times 100 \times 192$. Within the third block, the feature map changes from $100 \times 100 \times 192$ to $50 \times 50 \times 384$ and back to $100 \times 100 \times 192$. At last, features from all three blocks are concatenated as the final output, which has a size of $100 \times 100 \times 576$.

(3) Detection heads include one for large object classes (i.e. car, truck, trailer, bus, and construction vehicles) and one for small object classes (i.e. pedestrian, barrier, traffic cone, motorcycle, and bicycle). The large head takes the concatenated feature map from backbone network as input ($100 \times 100 \times 576$) while the small head takes the feature from the backbone's first convolutional stage as input ($200 \times 200 \times 96$). Each head contains a linear predictor for anchor box classification and a linear prediction for bounding box regression. The classification predictor outputs a confidence score for each anchor box and the regression predictor outputs adjustment coefficients (i.e. x, y, z, w, l, h, θ).

Loss function For classification, we adopt focal loss [16] and set $\alpha = 0.25$ and $\gamma = 2.0$. For regression output, we use smooth L1 loss (a.k.a. Huber loss) and set $\sigma = 3.0$, where σ controls where the transition between L1 and L2 happens. The final loss function is the classification loss multiplied by 2 plus the regression loss.

Training We train all of our models for 20 epochs and optimize using Adam [11] as the optimizer. We follow a learning rate schedule known as "one-cycle" [26]. The schedule consists of 2 phases. The first phase includes the first 40% training steps, during which we increase the learning rate from $\frac{0.003}{10}$ to 0.003 while decreasing the momentum from 0.95 to 0.85 following cosine annealing. The second phase includes the rest 60% training steps, during which we decrease the learning rate from 0.003 to $\frac{0.003}{10000}$ while increasing the momentum from 0.85 to 0.95. We use a fixed weight decay of 0.01.

Acknowledgements: This work was supported by the CMU Argo AI Center for Autonomous Vehicle Research.

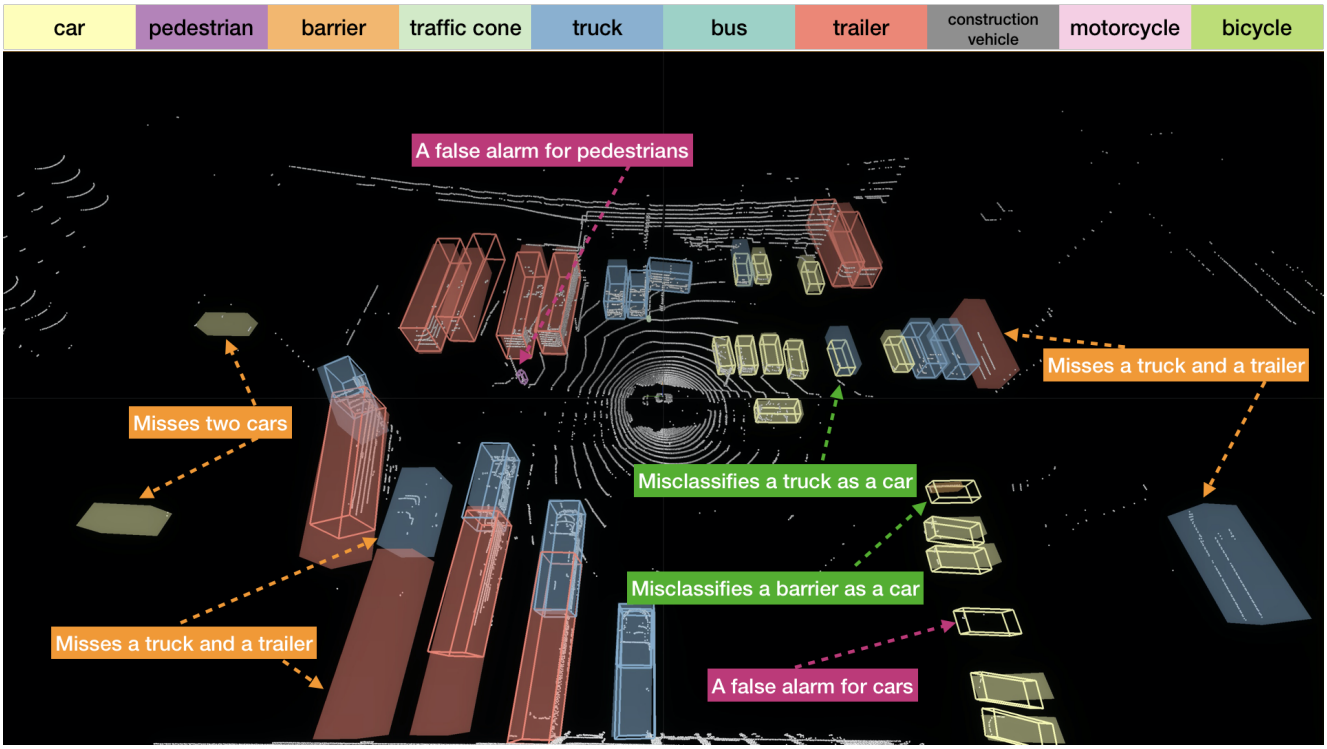


Figure A: We use an example frame (same as Fig. 6-(a)) from the video illustrate how we visualize. Solid transparent cuboids represent ground truth. Wireframe boxes represent predictions. Colors encode object classes (top). Inside this frame, our algorithm successfully detects most cars, trucks, and trailers. We also highlight all the mistakes made by our algorithm.

References

[1] John Amanatides and Andrew Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *EG 1987-Technical Papers*. Eurographics Association, 1987. 4

[2] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. 1

[3] Joachim Buhmann, Wolfram Burgard, Armin B Cremers, Dieter Fox, Thomas Hofmann, Frank E Schneider, Jiannis Strikos, and Sebastian Thrun. The mobile robot rhino. *AI Magazine*, 16(2):31–31, 1995. 2

[4] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. *arXiv preprint arXiv:1903.11027*, 2019. 4, 6, 8

[5] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. *arXiv preprint arXiv:1904.08755*, 2019. 3

[6] Andreas Eitel, Jost Tobias Springenberg, Luciano Spinello, Martin Riedmiller, and Wolfram Burgard. Multimodal deep learning for robust rgb-d object recognition. In *IROS*, pages 681–687. IEEE, 2015. 2

[7] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *IJRR*, 32(11):1231–1237, 2013. 6

[8] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous robots*, 34(3):189–206, 2013. 2, 4, 5

[9] Andrew E. Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *TPAMI*, 21(5):433–449, 1999. 2

[10] Eunyoung Kim and Gerard Medioni. 3d object recognition in range images using visibility context. In *IROS*, pages 3800–3807. IEEE, 2011. 2

[11] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 9

[12] Roman Klokov and Victor Lempitsky. Escape from cells: Deep kd-networks for the recognition of 3d point cloud models. In *ICCV*, pages 863–872, 2017. 2

[13] Jason Ku, Melissa Mozifian, Jungwook Lee, Ali Harakeh, and Steven L Waslander. Joint 3d proposal generation and object detection from view aggregation. In *IROS*, pages 1–8. IEEE, 2018. 2, 3

[14] Alex H Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *CVPR*, 2019. 2, 3, 4, 6, 9

[15] Yangyan Li, Rui Bu, Mingchao Sun, Wei Wu, Xinhan Di, and Baoquan Chen. Pointcnn: Convolution on x-transformed points. In *NeurIPS*, pages 820–830, 2018. 2

[16] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *CVPR*, pages 2980–2988, 2017. 9

[17] Wenjie Luo, Bin Yang, and Raquel Urtasun. Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net. In *CVPR*, pages 3569–3577, 2018. 3

[18] David Marr and Herbert Keith Nishihara. Representation and recognition of the spatial organization of three-dimensional shapes. *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 200(1140):269–294, 1978. 2

[19] Gregory P Meyer, Ankit Laddha, Eric Kee, Carlos Vallespi-Gonzalez, and Carl K Wellington. Lasernet: An efficient probabilistic 3d object detector for autonomous driving. In *CVPR*, pages 12677–12686, 2019. 1, 2

[20] Charles R Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J Guibas. Frustum pointnets for 3d object detection from rgb-d data. In *CVPR*, pages 918–927, 2018. 3

[21] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, pages 652–660, 2017. 1, 2

[22] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *NeurIPS*, pages 5099–5108, 2017. 2

[23] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3d recognition and pose using the viewpoint feature histogram. In *IROS*, pages 2155–2162. IEEE, 2010. 2

[24] Shaoshuai Shi, Xiaogang Wang, and Hongsheng Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *CVPR*, pages 770–779, 2019. 2, 3

[25] Martin Simon, Stefan Milz, Karl Amende, and Horst-Michael Gross. Complex-yolo: An euler-region-proposal for real-time 3d object detection on point clouds. In *ECCV*, pages 197–209. Springer, 2018. 2, 3

[26] Leslie N Smith. Cyclical learning rates for training neural networks. In *WACV*, pages 464–472. IEEE, 2017. 9

[27] Sebastian Thrun and Arno Bücken. Integrating grid-based and topological maps for mobile robot navigation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 944–951, 1996. 2

[28] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005. 4

[29] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *ACM TOG*, 38(5):146, 2019. 2

[30] Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *CVPR*, pages 9621–9630, 2019. 2

[31] Yan Yan, Yuxing Mao, and Bo Li. Second: Sparsely embedded convolutional detection. *Sensors*, 18(10):3337, 2018. 2, 3, 4, 9

[32] Bin Yang, Ming Liang, and Raquel Urtasun. Hdnet: Exploiting hd maps for 3d object detection. In *CoRL*, pages 146–155, 2018. 2, 3

[33] Theodore C Yapo, Charles V Stewart, and Richard J Radke. A probabilistic representation of lidar range data for efficient 3d object detection. In *CVPR Workshops*, pages 1–8. IEEE, 2008. 2

[34] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *CVPR*, pages 4490–4499, 2018. 1, 2, 3, 9