Supplementary Materials

1. Constructing 3DBev24k

3DBev24k is a novel dataset constructed by assembling objects and simulating Lidar physics with Blensor [1]. In what follows we describe the process of generating Lidar point clouds and annotations for 3DBev24k.

1.1. Simulated Data

Input: The simulation takes as input a manually constructed scene, which is primarily comprised of common beverage shapes, such as cans, bottles, and cases. The coordinate system of the 3D scenes is x (left/right), y (forward/back), and z (up/down). These are placed on 3D models of common shelf types that are found in many typical grocery stores. We manually build scenes to ensure that object placement is consistent with the laws of physics. The input scene is assumed to be "full", meaning that adding objects to the scene would result in object placements that are incongruent with physics (i.e., floating objects etc). Additionally, the objects are all placed in to "clusters". A cluster is defined as a group of objects that are placed in the same plane in the either the x or y dimension. Within a cluster, all objects are assumed to share the same class.

Generating Data: Given a "full" scene, we iteratively perturb the original scene with the following steps:

1. Choose number of vantage points. Vantage points are chosen in increments of 30 degrees. Scenes with one side will have 3 vantage points; scenes with 2 two sides will have 6 vantage points; scenes with 3 sides will have 9 vantage points, and scenes with 4 sides will have 12 vantage points.

2. Randomly perturb camera position and angle at each vantage point

3. Randomly assign geometric objects to a finegrained class, within a cluster. For each object type we look up a set of candidate classes and sample from the candidates with a predefined probability distribution. In nearly all cases we use a uniform distribution over candidate classes.

4. Randomly select objects, each with probability p, and remove them from the scene

5. We sample from a prior distribution over Lidar parameters (e.g., point density, noise) and simulate Lidar physics [1].

6. Write labels: semantic segmentation, bounding boxes, and counts

7. Restore scene to original state

See Figure 4 for an illustration of the simulator as it progresses.

Output: The simulation outputs a variety of annotations including bounding boxes, semantic segmentation labels, and raw counts vectors. Additionally, we output the generated point cloud to a file.

2. Training Details

In this section we describe settings used during training.

2.1. Preprocessing

We scale each scene's point cloud to lie in the unit ball by subtracting each point from the mean, and dividing each point by the largest euclidean distance from the origin.

2.2. Network Architectures

2.2.1 CountNet3D

CountNet3D consists of a PointNet backbone and a fully connected regression module. Our PointNet backbone is built with 8 input channels and 4 linear layers with sizes [64, 128, 256, 512]. Each layer uses batch norm and relu activations. We apply a max pooling layer to produce a 512 dimensional global feature tensor. We have 21 geometric classes, producing a one hot tensor of 21 dimensions. These two are concatenated into a 533 input tensor to the regression module. The regressor is a 5 layer, full-connected network with dimensions [512, 256, 64, 64, 64].

2.2.2 YOLOv5

We train a YOLOv5 model for our 2D detector in the image layer. The model consists of 407 layers and 108,547,000 parameters

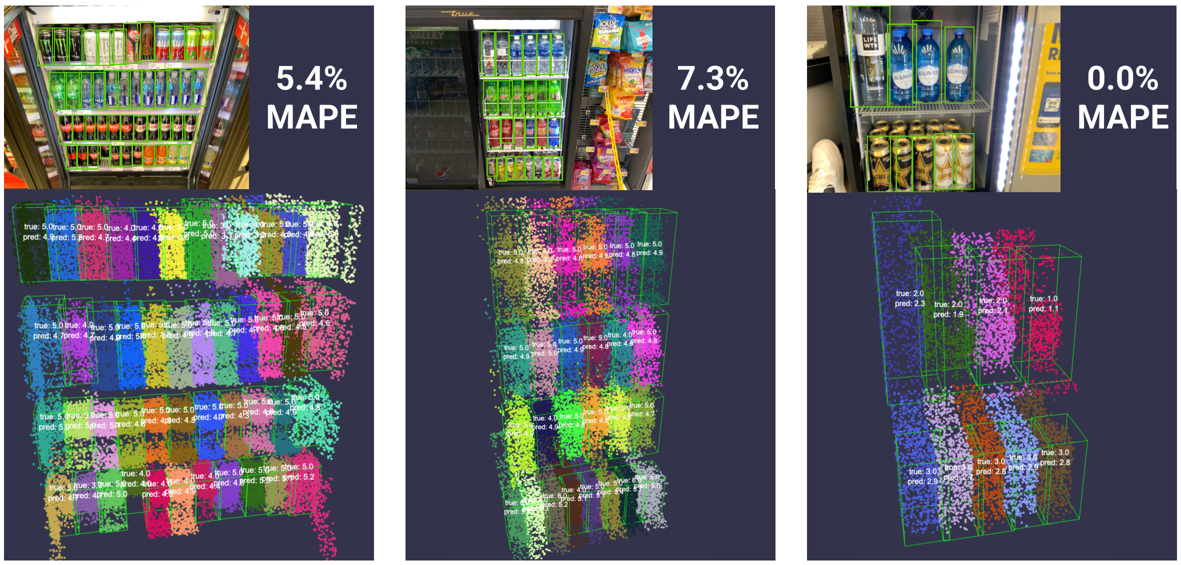


Figure 1: **Example predictions from CountNet3D on real-world data**. (Best viewed in color with zoom in). For each scene we show the image, the PointBeam proposals, the ground truth, and the predicted counts. We also display the scene-level MAPE (after rounding). Each beam is given a randomly generated color to highlight the beam regions. We only display the global $\langle x, y, z \rangle$ for visualization clarity. We observe that under extreme occlusion, where both 2D and 3D object detectors are likely to fail, CountNet3D is able to accurately predict the object counts.

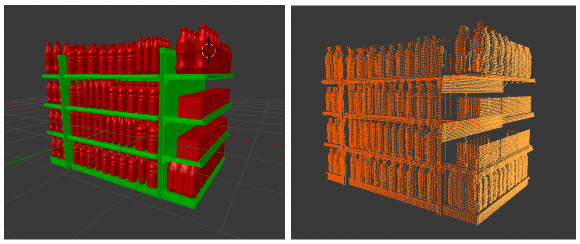


Figure 2: An example scene from 3DBev24k. We build a 3D model of the scene (left), and simulate the lidar physics with BlenSor [1] (right) while also randomizing camera vantage points, object poses, and noise parameters.

2.3. Hardware & Software

All models are implemented in PyTorch and trained using four NVIDIA GeForce RTX 2080's or four NVIDIA GeForce RTX 3070's. The PointNet backbone used in our experiment was implemented by [3]. Our YOLO implementation was provided by [2].

2.4. YOLO Training and Evaluation

We report our evaluation statistics for our YOLOv5 model on the real-world dataset. While the dataset only contains 3D scans of 359 finegrained classes, the YOLOv5 model is trained on a much larger dataset covering 3,600 classes. The set of finegrained classes used is a proper subset of the 3,600 seen during YOLO training.

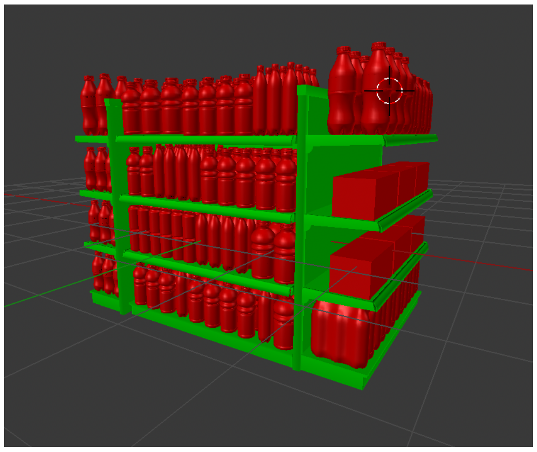
Table 1: YOLO evaluation

Precision	Recall	mAP@.5
0.615	0.6282	0.5581

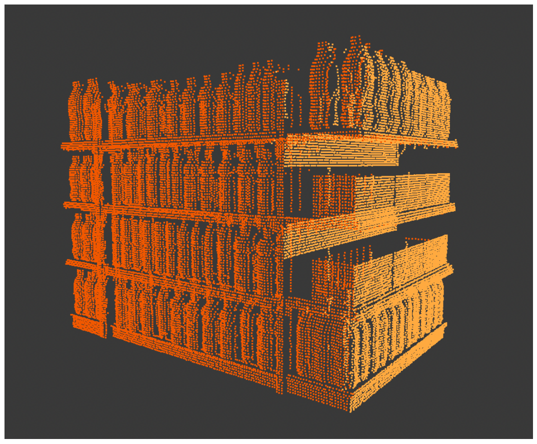
3. Geometry Dictionary

We create a curated dictionary that maps a finegrained class to a canonical geometric types. The geometric types are:

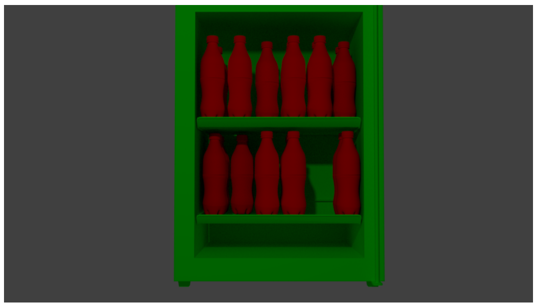
- twoliter
- 20ozGatoradebottle
- 12ozSlimcan
- 2Oozbottle
- 17ozFlatsidebottle
- 16ozcan
- twelvepack
- 8-12ozCanbox
- 4-16ozCanbox
- sixpack



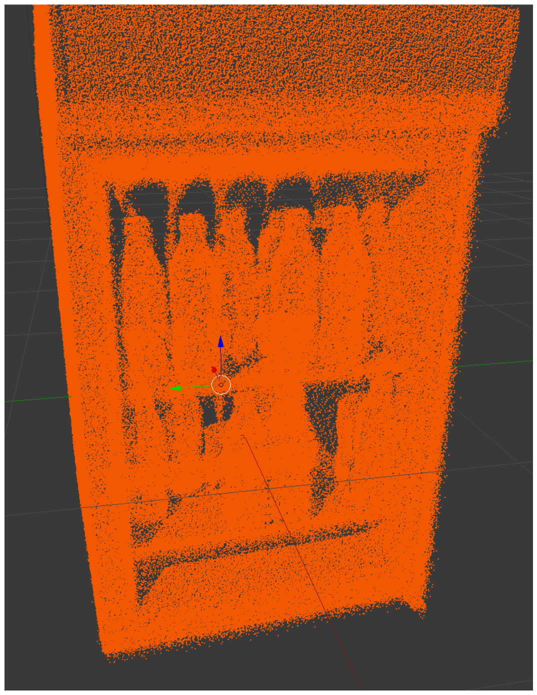
(a) Scene 1: 3D Model



(b) Scene 1: Point Cloud



(c) Scene 2: 3D Model



(d) Scene 2: Point Cloud

Figure 3: We show two scenes from 3DBev24k. Above we display a screen capture of the 3D model and an image rendering of the simulated point cloud [1]

- 32ozGatoradebottle
- 24-12ozCanbox
- 6-17ozWrappedflatsidebottle
- 1lBottle
- 16ozIsotonicbottle
- 6-7pt5ozCan
- 12-17ozWrappedflatsidebottle
- 8-20ozGatoradebottle
- 24-17ozWrappedflatsidebottle
- other

A finegrained class describes the semantic class, which includes both visual and shape semantics. In our retail dataset, this is specified down to the unique product identifier (e.g., 049000000443) For example, one finegrained class might be "coca_cola_20oz_049000000443".

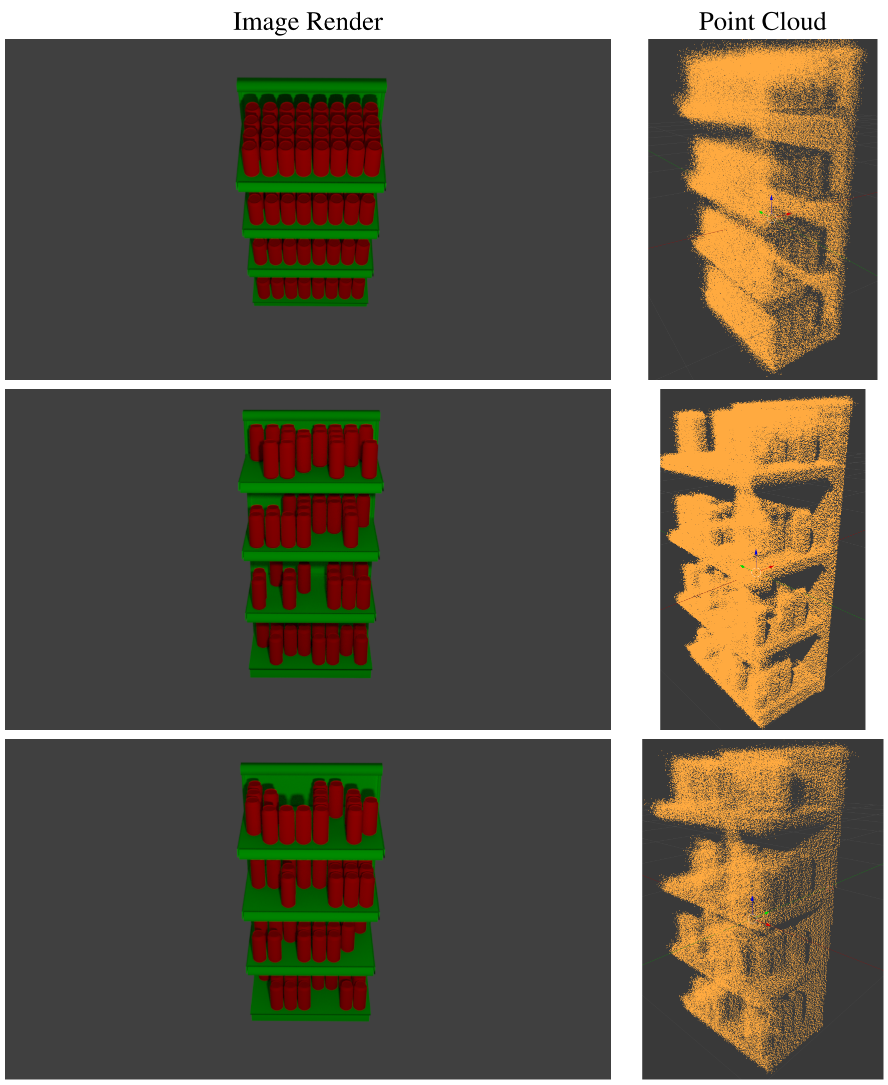


Figure 4: Instances of our simulated dataset. We display an image rendering of the scene (left) along with the simulated point cloud (right). Target objects are red and the background is green to facilitate automatic semantic segmentation labelling. (Top) A “full” scene, or the first iteration of the simulator over this scene type. (Middle) A subsequent iteration, where some objects are removed. (Bottom) The final iteration of the scene after additional perturbation. Note that the noise and density of the point clouds is varied, in addition to the vantage point of the camera.

This finegrained class would be mapped “coca_cola_20oz_049000000443” $\rightarrow$ ‘20Oozbottle”. A second example might be “monster_energy_16oz_070847811169” $\rightarrow$ “16ozcan”.

The geometry dictionary does require some manual engineering, but is an important dimensionality step for large taxonomies. If the number of finegrained classes is small, this step could be skipped and a one hot encoding could be computed directly from finegrained classes.

4. Matching Global 3D detections to Finegrained Classes

We compare the proposed CountNet3D to state-of-the-art 3D object detectors trained on the global point cloud. Because 3D object detectors primarily look for lower resolution, 3D shapes we need a way to match the 3D detections to more finegrained classes. For the results in Table 2 take as input `3D`, as opposed to `RGB+3D`, we use a closest point algorithm to match the 3D detections and a finegrained class.

The primary goal of the closest point matching algorithm was to label the unlabeled points from the output of the 3D

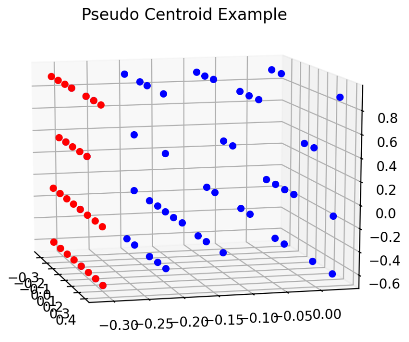


Figure 5: Example of the closest matching algorithm. Red points are centroids from objects that were detected from the image layer, whose class is given. Blue points are objects detected by the 3D object detector; their class must be inferred from the red points

object detection model. Using the already classified centroid from the images we mapped their labels to the unlabeled 3D detections. With the centroids and the objects matched, we were able to count all of the detected objects by class.
The algorithm uses the euclidean distance.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \quad (1)$$

where $\boldsymbol{x_1}$= the centroid (the labeled point) and, $\boldsymbol{x_2}$= the unlabeled point.

Due to the fact that the 2D detections lifted to 3D are not guaranteed to lie on a constant plane, auto generated centroids, pseudo centroids, were created to ensure equal distancing between 2D detections and unlabeled points. (see the figure bellow). These pseudo centroids are generated by first finding the closest point to the camera in the x direction, and then generating a centroid for each row all sharing the same x direction.

In Figure 5 we show an example of the closest point matching algorithm. As you can see the new centroids are positioned in the lowest x value. This ensures that the only points that are mapped together, are points in the same row.

5. Hyperparameter study

We perform a small set of experiments studying the impact of the maximum depth parameter, δ, on the performance of CountNet3D. This parameter controls the size, and consequently the quality of the PointBeam proposals. In general, we see that small δ values cause the performance to degrade rapidly. This has the effect that important points

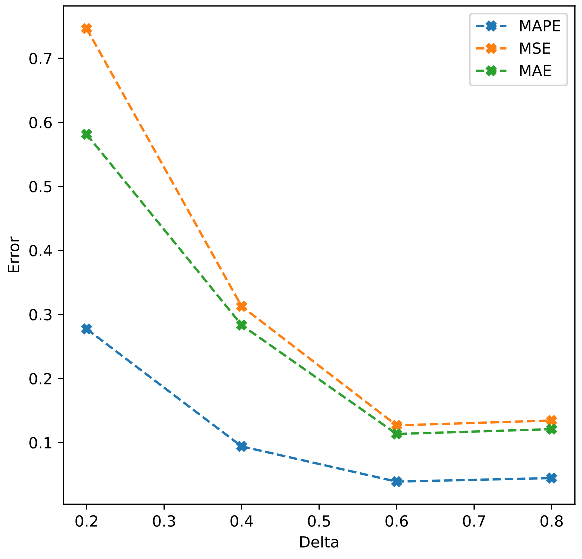


Figure 6: The impact of the maximum depth parameter, δ, on MAPE, MSE, and MAE. We see that small δ values harm the accuracy. As δ approaches 1, performances improves.

which penetrated depth-wise into the occluded scene are now filtered out. When $\delta = 0.2$, the PointBeams roughly only cover the front-most objects and performance is similar to the detection-based methods. As δ increases, performance increases as more points are included in the PointBeam proposal. In our main experiments, we set $\delta = 0.6$.

References

[1] Michael Gschwandtner, Roland Kwitt, Andreas Uhl, and Wolfgang Pree. Blensor: Blender sensor simulation toolbox. In *Proceedings of the 7th International Conference on Advances in Visual Computing - Volume Part II*, ISVC'11, page 199–208, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Glenn Jocher, Alex Stoken, Ayush Chaurasia, Jirka Borovec, NanoCode012, TaoXie, Yonghye Kwon, Kalen Michael, Liu Changyu, Jiacong Fang, Abhiram V, Laughing, tkianai, yxNONG, Piotr Skalski, Adam Hogan, Jebastin Nadar, imyhxy, Lorenzo Mammana, AlexWang1900, Cristi Fati, Diego Montes, Jan Hajek, Laurentiu Diaconu, Mai Thanh Minh, Marc, albinxavi, fatih, oleg, and wanghaoyang0106. ultralytics/yolov5: v6.0 - YOLOv5n 'Nano' models, Roboflow integration, TensorFlow export, OpenCV DNN support, Oct. 2021.

[3] Clement Fuji Tsang, Masha Shugrina, Jean Francois Lafleche, Towaki Takikawa, Jiehan Wang, Charles

Loop, Wenzheng Chen, Krishna Murthy Jatavallabhula, Edward Smith, Artem Rozantsev, Or Perel, Frank Shen, Jun Gao, Sanja Fidler, Gavriel State, Jason Gorski, Tommy Xiang, Jianing Li, Michael Li, and Rev Lebaredian. Kaolin, 2019.