In [1]:
```
from IPython.display import HTML
HTML('<center><iframe width="700" height="400" src="https://www.youtube.com/embed/t
```

Out[1]:

What is LiDAR?

LiDAR (Light Detection and Ranging) is a method used to generate accurate 3D representations of the surroundings, and it uses laser light to achieve this. Basically, the 3D target is illuminated with a laser light (a focused, directed beam of light) and the reflected light is collected by sensors. The time required for the light to reflect back to the sensor is calculated.

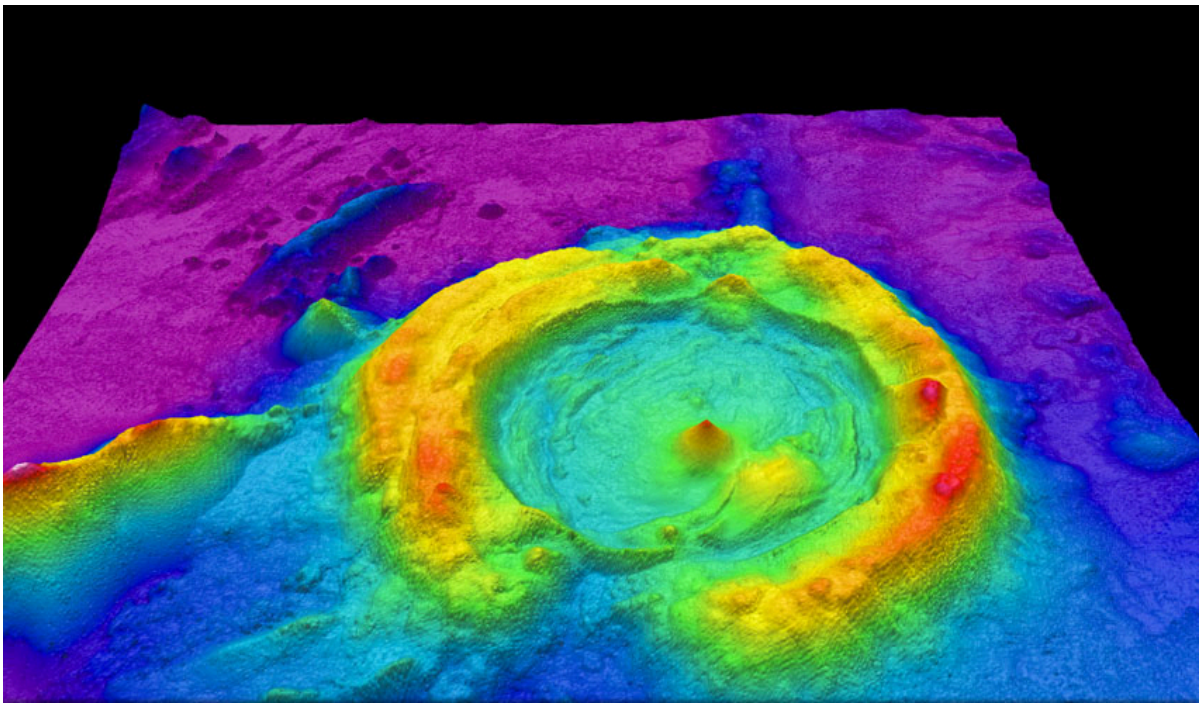
Different sensors collect light from different parts of the object, and the times recorded by the sensors would be different. This difference in time calculated by the sensors can be used to calculate the depth of the object. This depth information combined with the 2D represenation of the image provides an accurate 3D representation of the object. This process is similar to actual human vision. Two eyes make observations in 2D and these two pieces of information are combined to form a 3D map (depth perception). This is how humans sense the world around us.

This technology is used to create 3D representations in many real world scenarios. For example, it is used in farms to help sow seeds and remove weeds. A moving robot uses LiDAR to to create a 3D map of its surroundings and using this map, it avoids obstacles and completes its tasks. This technology is also used in archaeology. LiDAR is used to create 3D renderings of 2D scans of artifacts. This gives an accurate idea of the 3D shape of the artifact when the artifact cannot be excavated for whatever reason. Finally, LiDAR can also be used to render high quality 3D maps of ocean floors and other inaccesible terrains, making it very

useful to geologists and oceanographers. Below is a 3D map of an ocean floor generated using LiDAR:

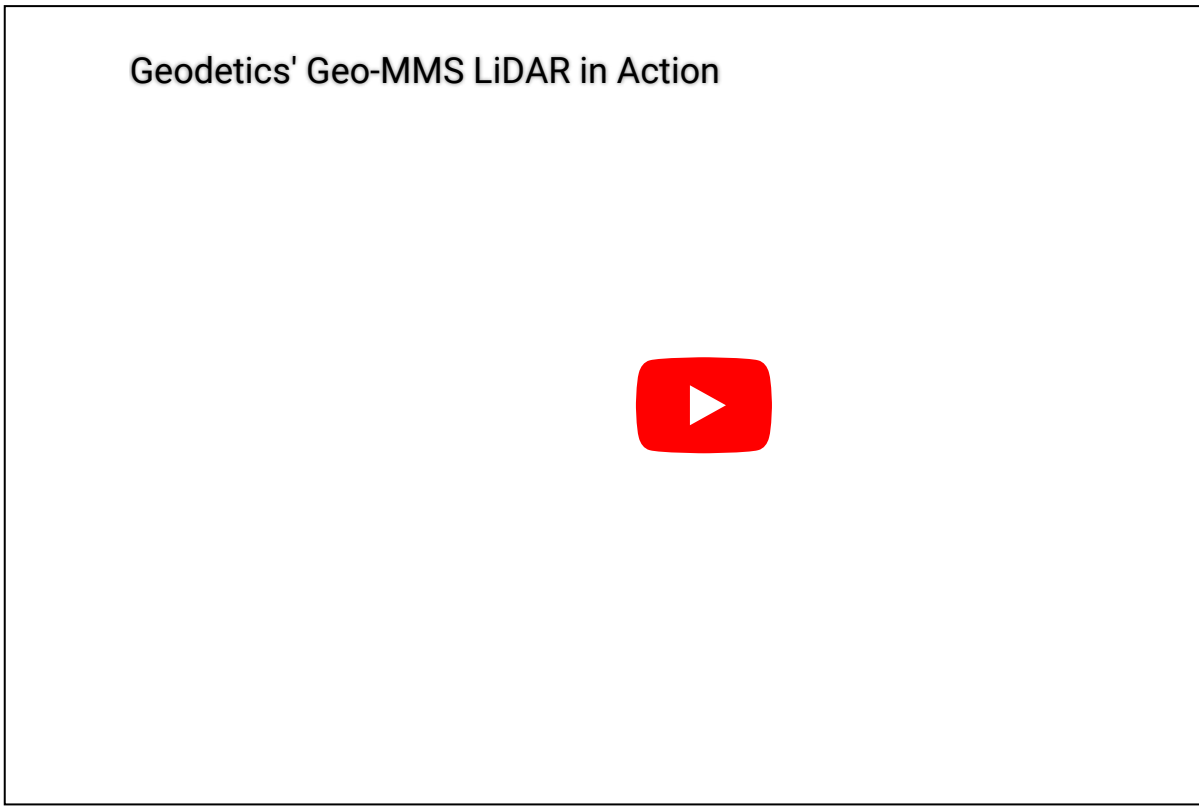
In [2]:
```
from IPython.display import Image
image_path = "mine.jpg"
Image(filename=image_path)
```

Out[2]:



In [3]:
```
HTML('<center><iframe width="700" height="400" src="https://www.youtube.com/embed/x
```

Out[3]:



In [4]:
```
from IPython.display import Image

# Specify the path to your GIF file
gif_path = "mee.gif"
```

```
# Display the GIF
Image(url=gif_path)
```

Out[4]: 

The above GIF roughly demonstrates how LiDAR works. Basically, laser beams are shot in all directions by a laser. The laser beams reflect off the objects in their path and the reflected beams are collected by a sensor. Now, a special device called a Flash LiDAR Camera is used to create 3D maps using the information from these sensors.

Flash LiDAR Camera

In [5]:
```
from IPython.display import Image
image_path = "lidar.jpg"
Image(filename=image_path)
```

Out[5]:

The device featured in the image above is called a Flash LiDAR Camera. The focal plane of a Flash LiDAR camera has rows and columns of pixels with ample "depth" and "intensity" to create 3D landscape models. Each pixel records the time it takes each laser pulse to hit the target and return to the sensor, as well as the depth, location, and reflective intensity of the object being contacted by the laser pulse.

The Flash LiDAR uses a single light source that illuminates the field of view in a single pulse. Just like a camera that takes pictures of distance, instead of colors.

The onboard source of illumination makes Flash lidar an active sensor. The signal that is returned is processed by embedded algorithms to produce a nearly instantaneous 3D rendering of objects and terrain features within the field of view of the sensor. The laser

pulse repetition frequency is sufficient for generating 3D videos with high resolution and accuracy. The high frame rate of the sensor makes it a useful tool for a variety of applications that benefit from real-time visualization, such as autonomous vehicle driving. By immediately returning a 3D elevation mesh of target landscapes, a flash sensor can be used by an autonomous vehicle to make decisions regarding speed adjustment, braking, steering, etc.

This type of camera is attached to the top of autonomous cars, and these cars use this to navigate while driving.

Now, since it is clear what LiDAR is and how it works, we can get right to visualizing the dataset.

In [6]:
```
import os
import gc
import numpy as np
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

import json
import math
import sys
import time
from datetime import datetime
from typing import Tuple, List

import cv2
import matplotlib.pyplot as plt
import sklearn.metrics
from PIL import Image

from matplotlib.axes import Axes
from matplotlib import animation, rc
import plotly.graph_objs as go
import plotly.tools as tls
from plotly.offline import plot, init_notebook_mode
import plotly.figure_factory as ff

init_notebook_mode(connected=True)

import seaborn as sns
from pyquaternion import Quaternion
from tqdm import tqdm

from lyft_dataset_sdk.utils.map_mask import MapMask
from lyft_dataset_sdk.lyftdataset import LyftDataset
from lyft_dataset_sdk.utils.geometry_utils import view_points, box_in_image, BoxVis
from lyft_dataset_sdk.utils.geometry_utils import view_points, transform_matrix
from pathlib import Path

import struct
from abc import ABC, abstractmethod
from functools import reduce
from typing import Tuple, List, Dict
import copy
```

In [7]:
```
DATA_PATH = 'C:/Users/kalsa/Downloads/3d-object-detection-for-autonomous-vehicles/'
```

In [8]:
```
train = pd.read_csv(DATA_PATH + 'train.csv')
sample_submission = pd.read_csv(DATA_PATH + 'sample_submission.csv')
```

In [9]:
```
train.describe()
```

Out[9]:

	Id	PredictionString
count	22680	22680
unique	22680	22680
top	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	2680.2830359778527 698.1969292852777 -18.04776...
freq	1	1

In [10]:
```
sample_submission.describe()
```

Out[10]:

	PredictionString
count	0.0
mean	NaN
std	NaN
min	NaN
25%	NaN
50%	NaN
75%	NaN
max	NaN

In [11]:
```
object_columns = ['sample_id', 'object_id', 'center_x', 'center_y', 'center_z',
                  'width', 'length', 'height', 'yaw', 'class_name']
objects = []
for sample_id, ps in tqdm(train.values[:]):
    object_params = ps.split()
    n_objects = len(object_params)
    for i in range(n_objects // 8):
        x, y, z, w, l, h, yaw, c = tuple(object_params[i * 8: (i + 1) * 8])
        objects.append([sample_id, i, x, y, z, w, l, h, yaw, c])
train_objects = pd.DataFrame(
    objects,
    columns = object_columns
)
```

```
100%|██████████████████████████████████████████████████████████████████████| 2
2680/22680 [00:03<00:00, 7297.13it/s]
```

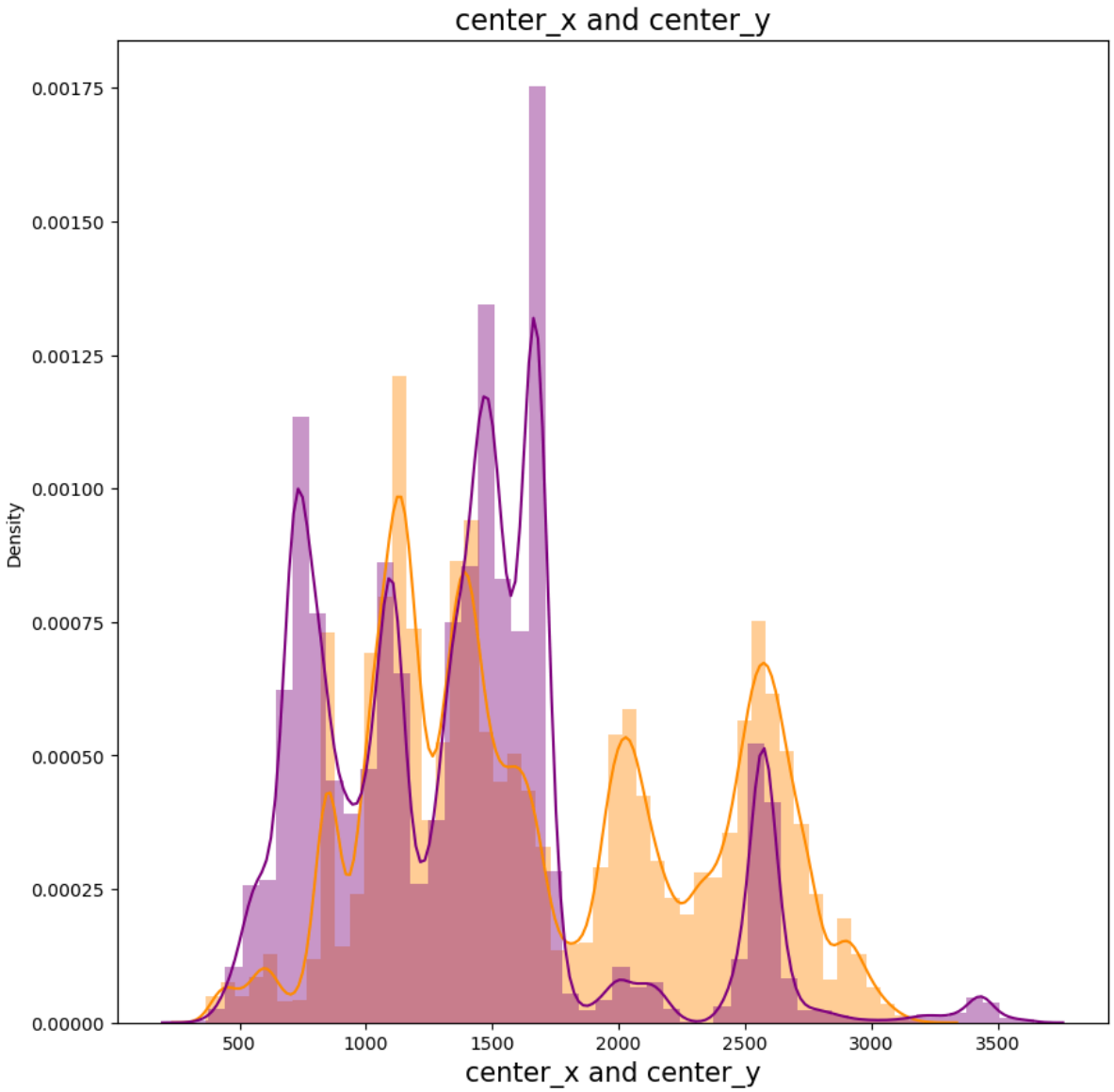
In [12]:
```
numerical_cols = ['object_id', 'center_x', 'center_y', 'center_z', 'width', 'length
train_objects[numerical_cols] = np.float32(train_objects[numerical_cols].values)
```

In [13]:
```
train_objects.head()
```

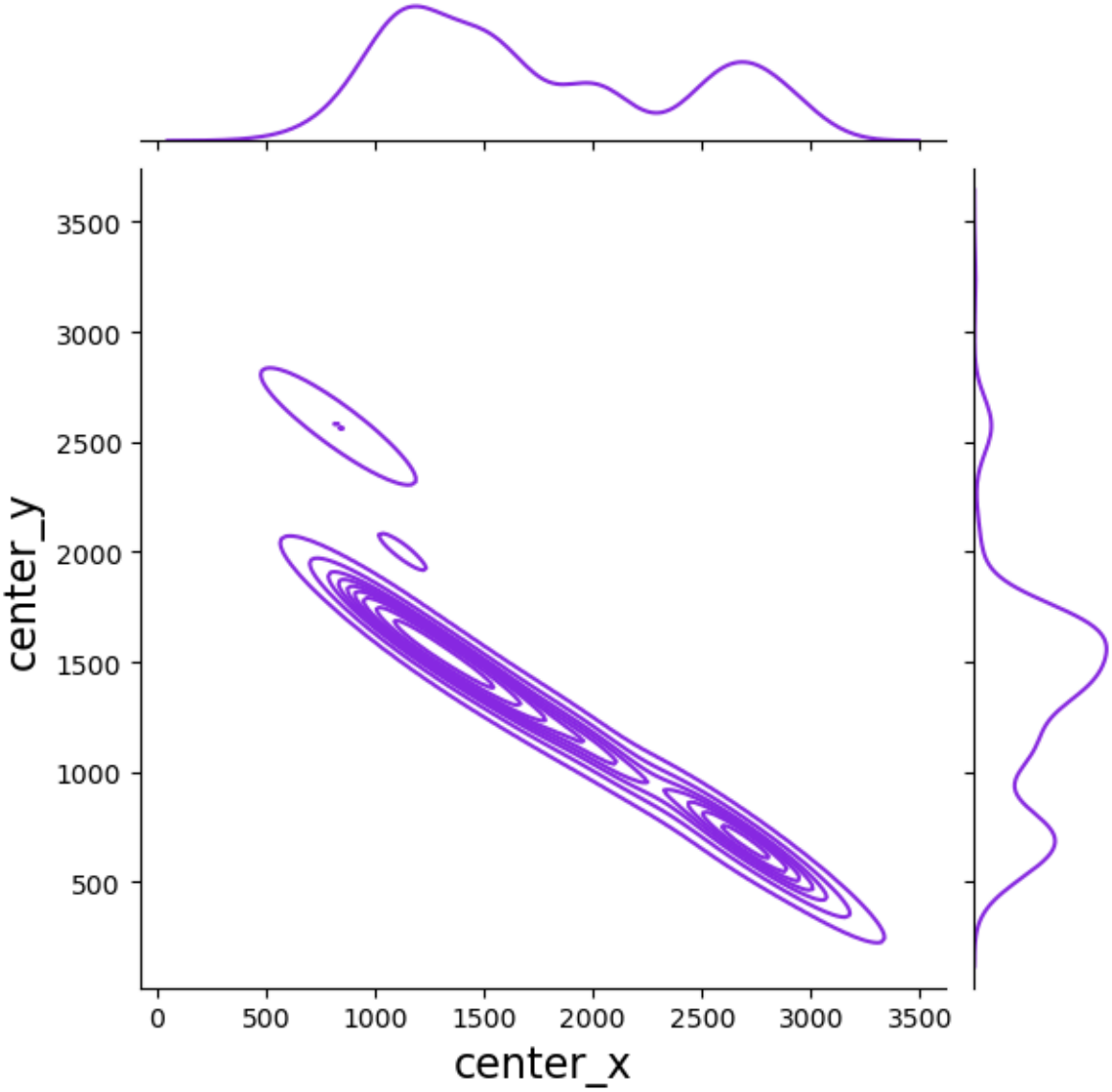
Out[13]:

	sample_id	object_id	center_x	center_y	cente
0	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	0.0	2680.282959	698.196899	-18.0477
1	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	1.0	2691.997559	660.801636	-18.6742
2	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	2.0	2713.607422	694.403503	-18.5899
3	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	3.0	2679.986816	706.910156	-18.3495
4	db8b47bd4ebdf3b3fb21598bb41bd8853d12f8d2ef25ce...	4.0	2659.352051	719.417480	-18.4429

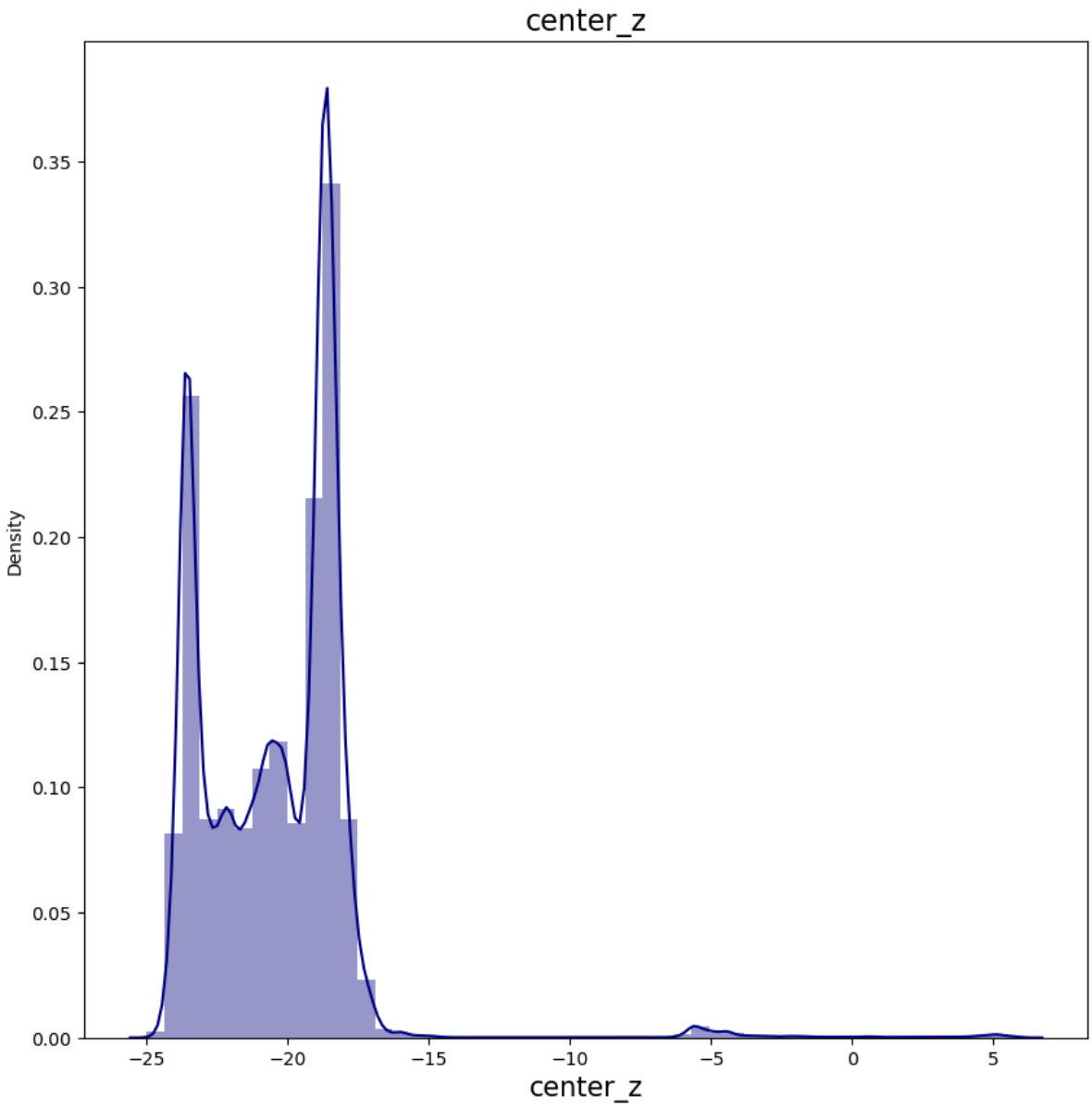
In [14]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['center_x'], color='darkorange', ax=ax).set_title('cente
sns.distplot(train_objects['center_y'], color='purple', ax=ax).set_title('center_x
plt.xlabel('center_x and center_y', fontsize=15)
plt.show()
```



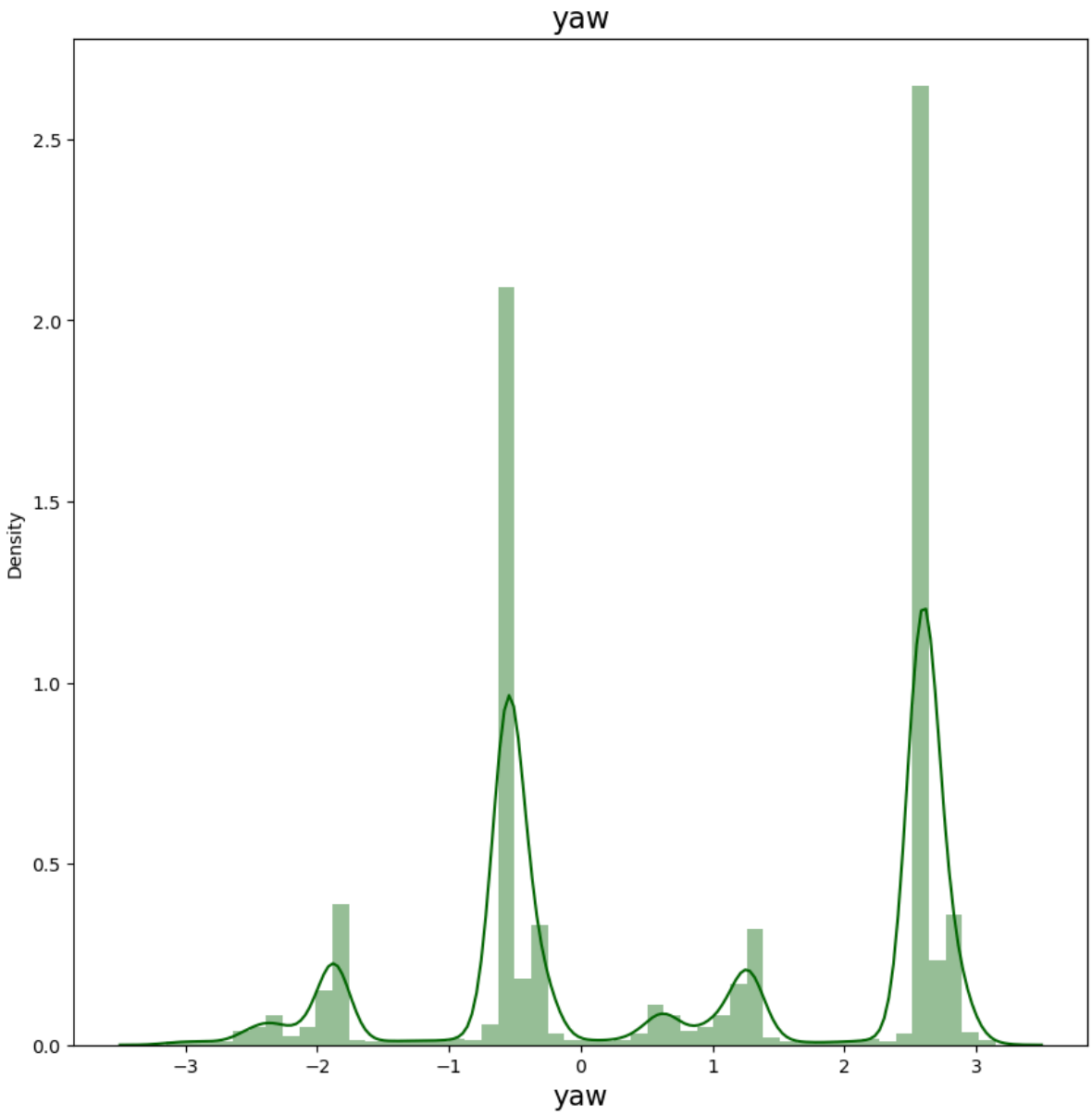
In [15]:
```
new_train_objects = train_objects.query('class_name == "car"')
plot = sns.jointplot(x=new_train_objects['center_x'][:1000], y=new_train_objects['c
plot.set_axis_labels('center_x', 'center_y', fontsize=16)
plt.show()
```



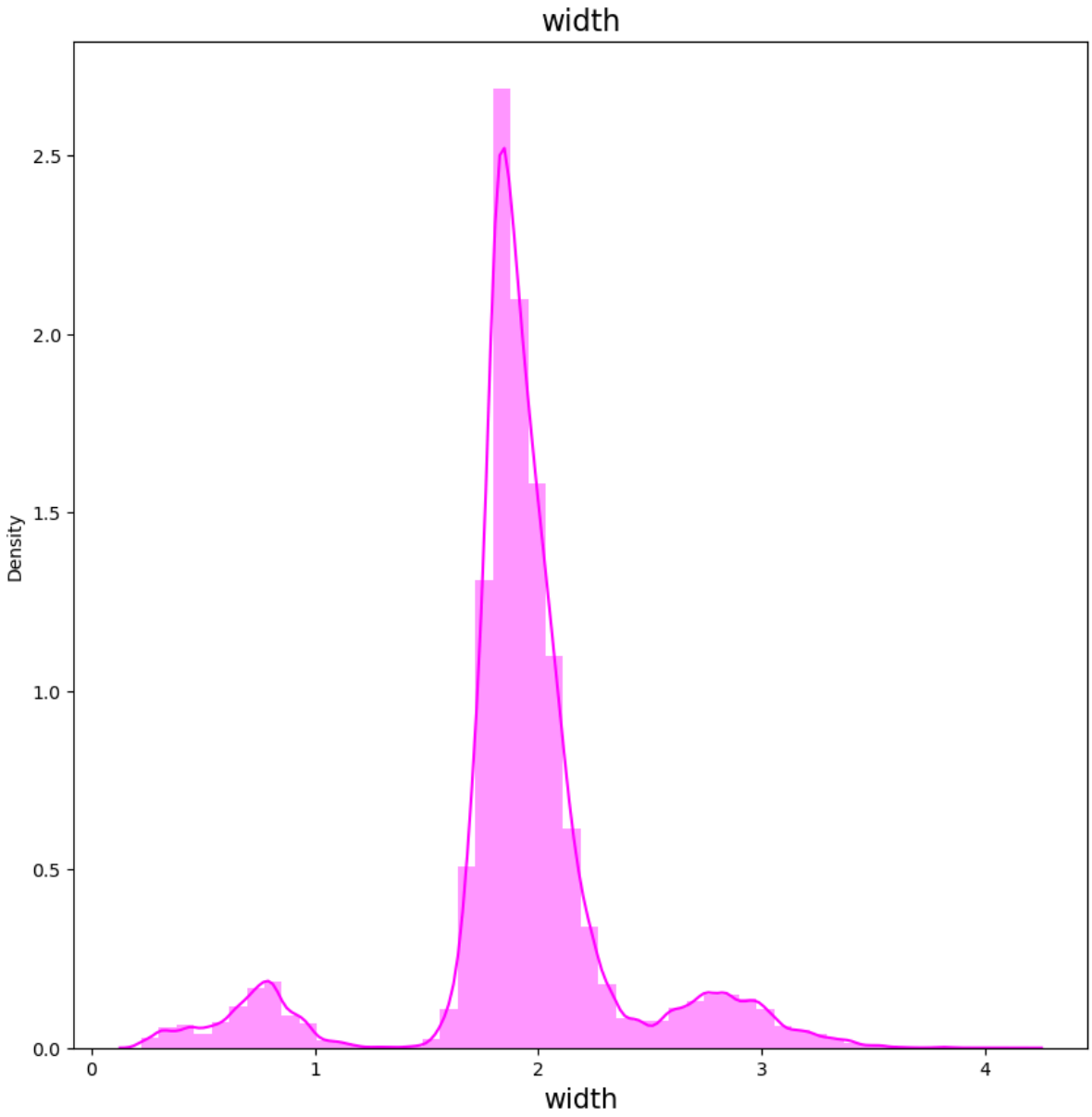
In [16]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['center_z'], color='navy', ax=ax).set_title('center_z',
plt.xlabel('center_z', fontsize=15)
plt.show()
```



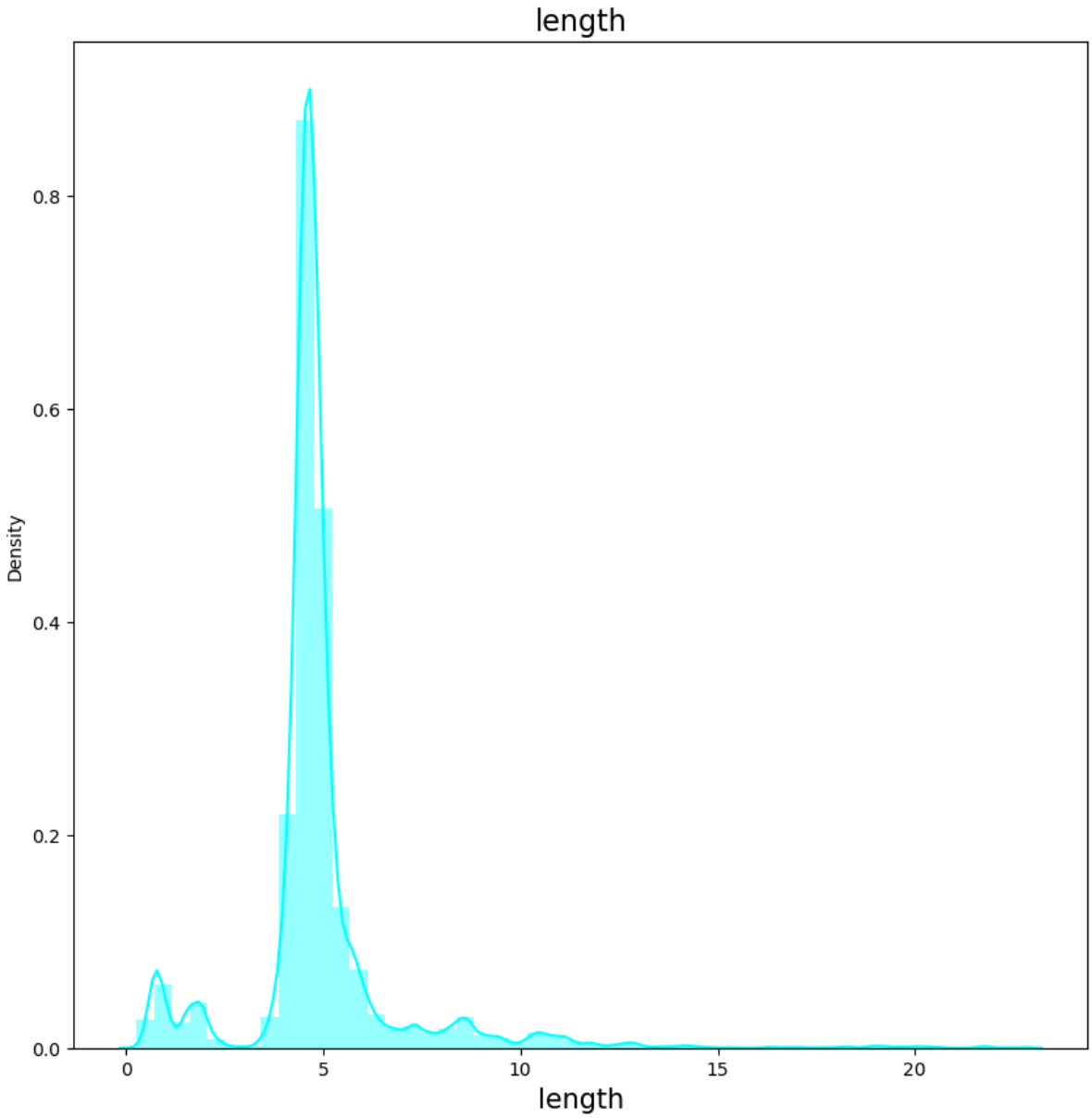
In [17]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['yaw'], color='darkgreen', ax=ax).set_title('yaw', fonts
plt.xlabel('yaw', fontsize=15)
plt.show()
```



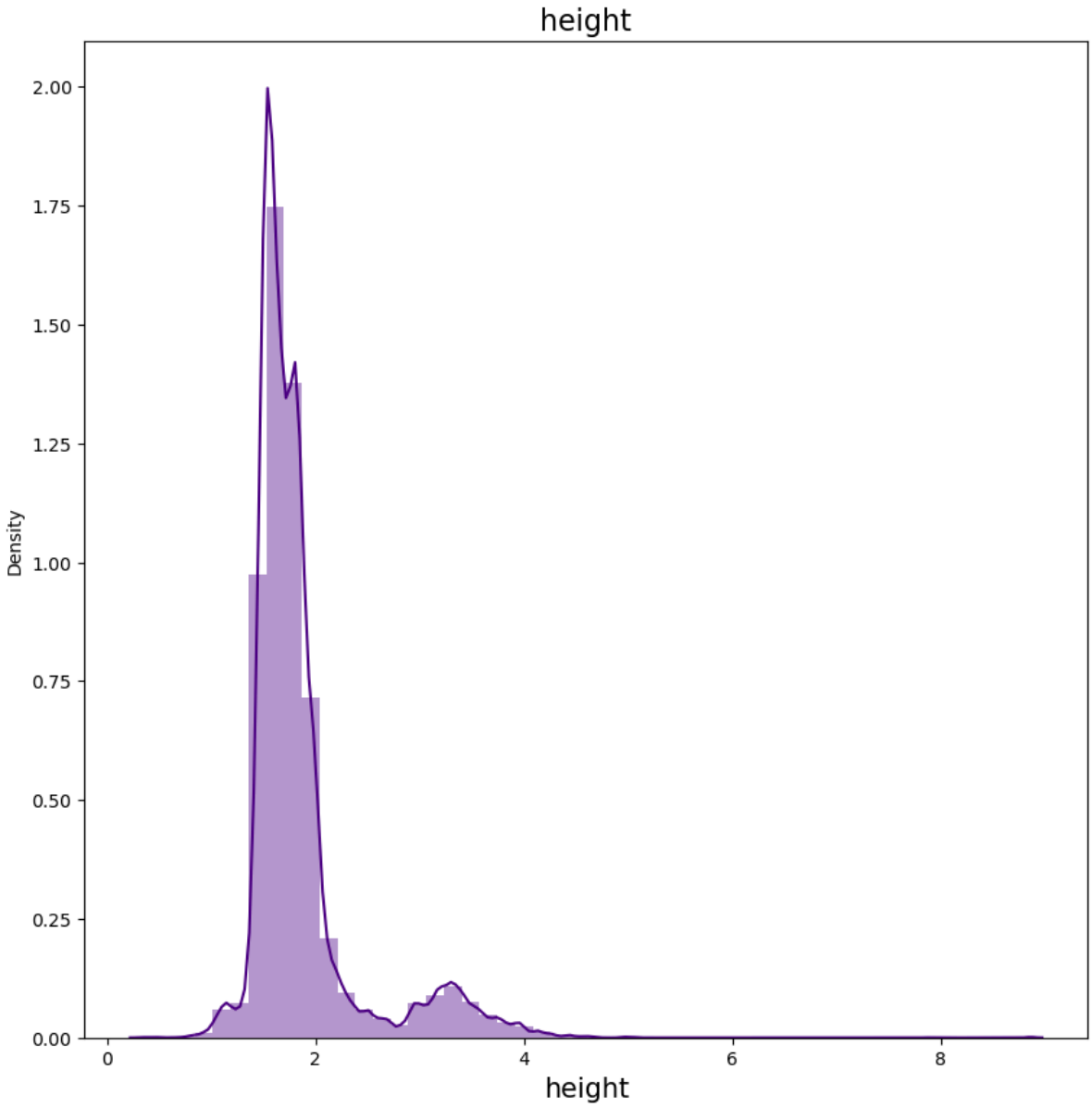
In [18]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['width'], color='magenta', ax=ax).set_title('width', for
plt.xlabel('width', fontsize=15)
plt.show()
```



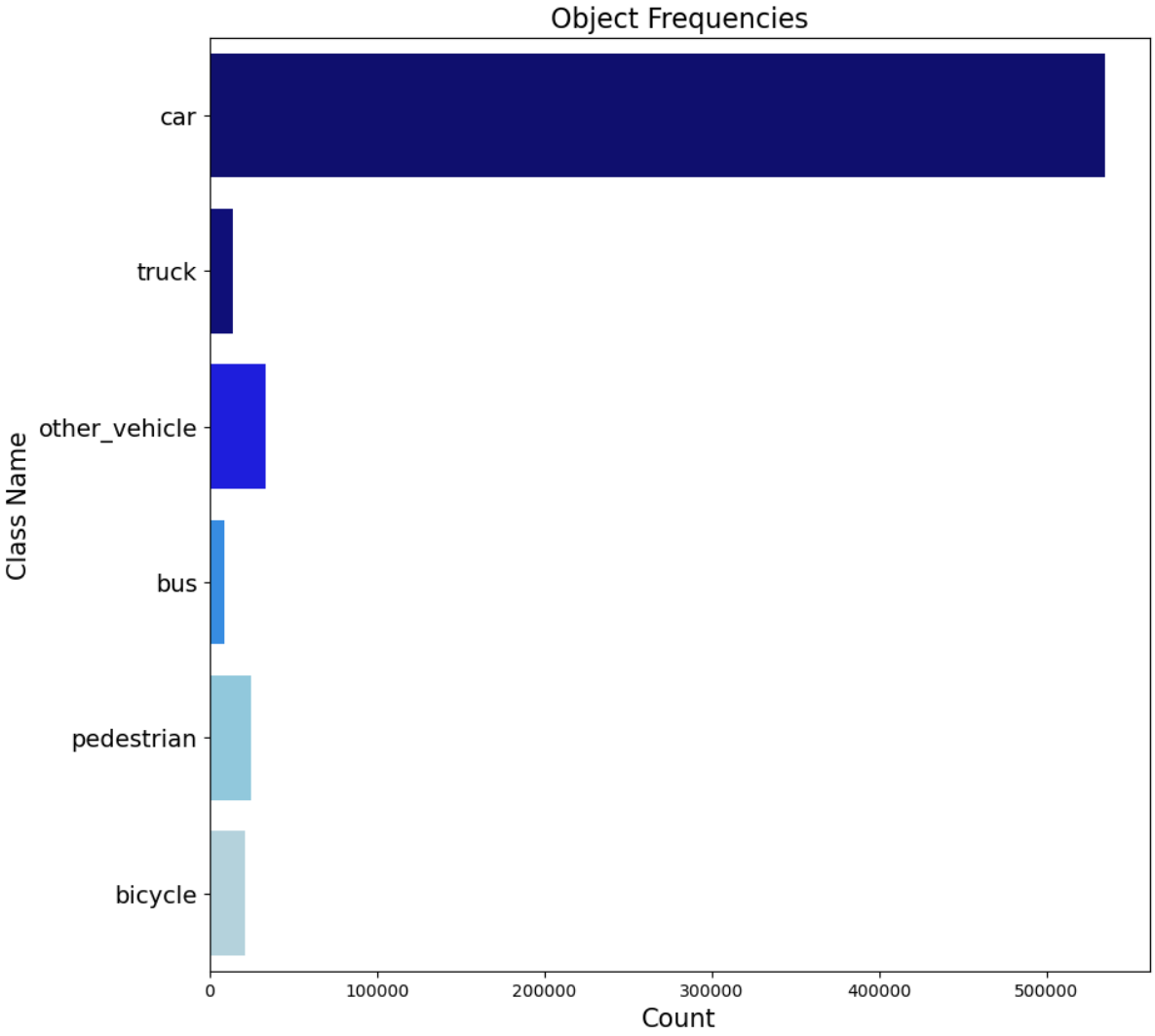
In [19]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['length'], color='cyan', ax=ax).set_title('length', font
plt.xlabel('length', fontsize=15)
plt.show()
```



In [20]:
```
fig, ax = plt.subplots(figsize=(10, 10))
sns.distplot(train_objects['height'], color='indigo', ax=ax).set_title('height', fo
plt.xlabel('height', fontsize=15)
plt.show()
```

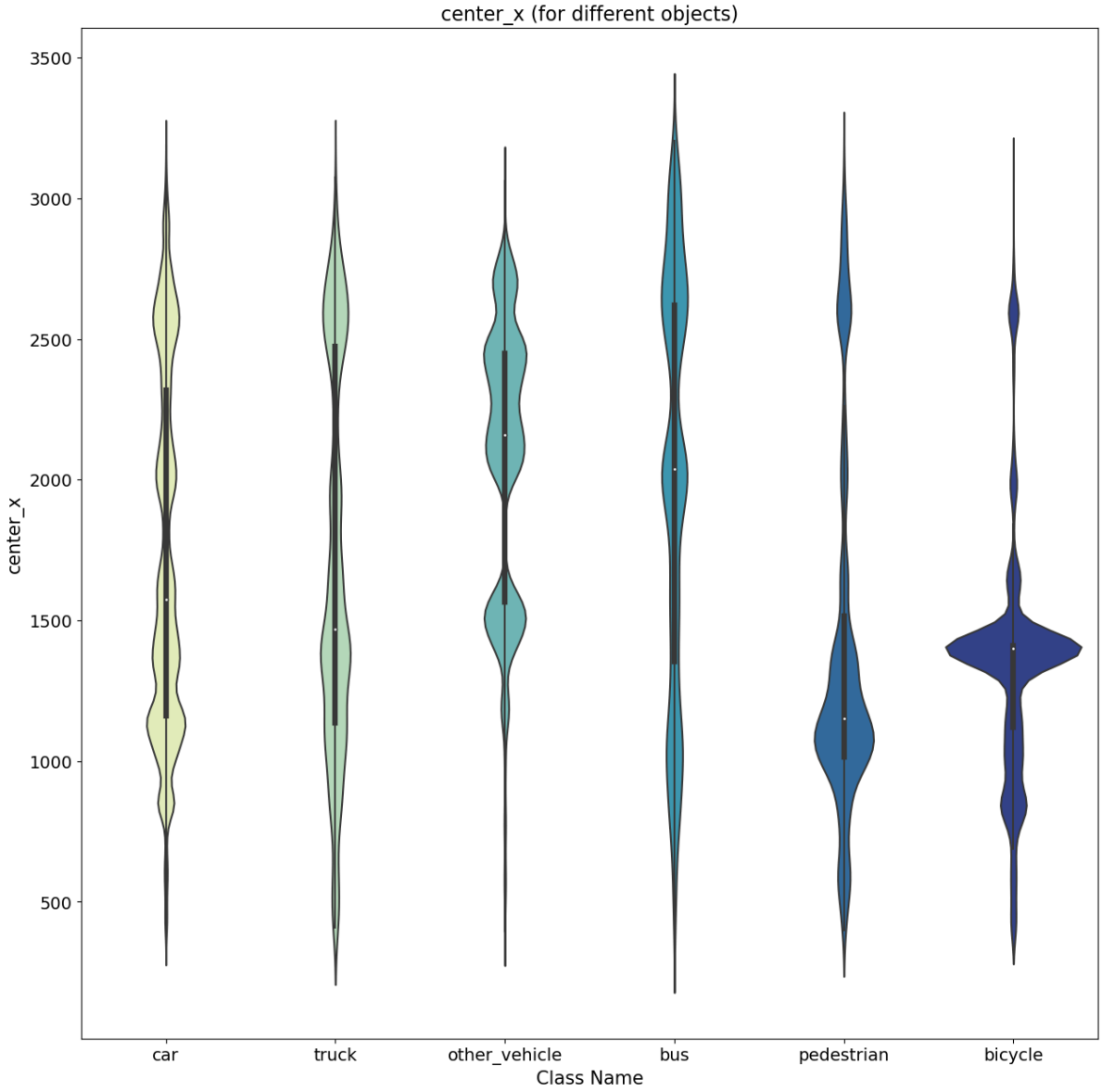


In [21]:
```
fig, ax = plt.subplots(figsize=(10, 10))
plot = sns.countplot(y="class_name", data=train_objects.query('class_name != "motor
                     palette=['navy', 'darkblue', 'blue', 'dodgerblue', 'skyblue',
plt.yticks(fontsize=14)
plt.xlabel("Count", fontsize=15)
plt.ylabel("Class Name", fontsize=15)
plt.show(plot)
```

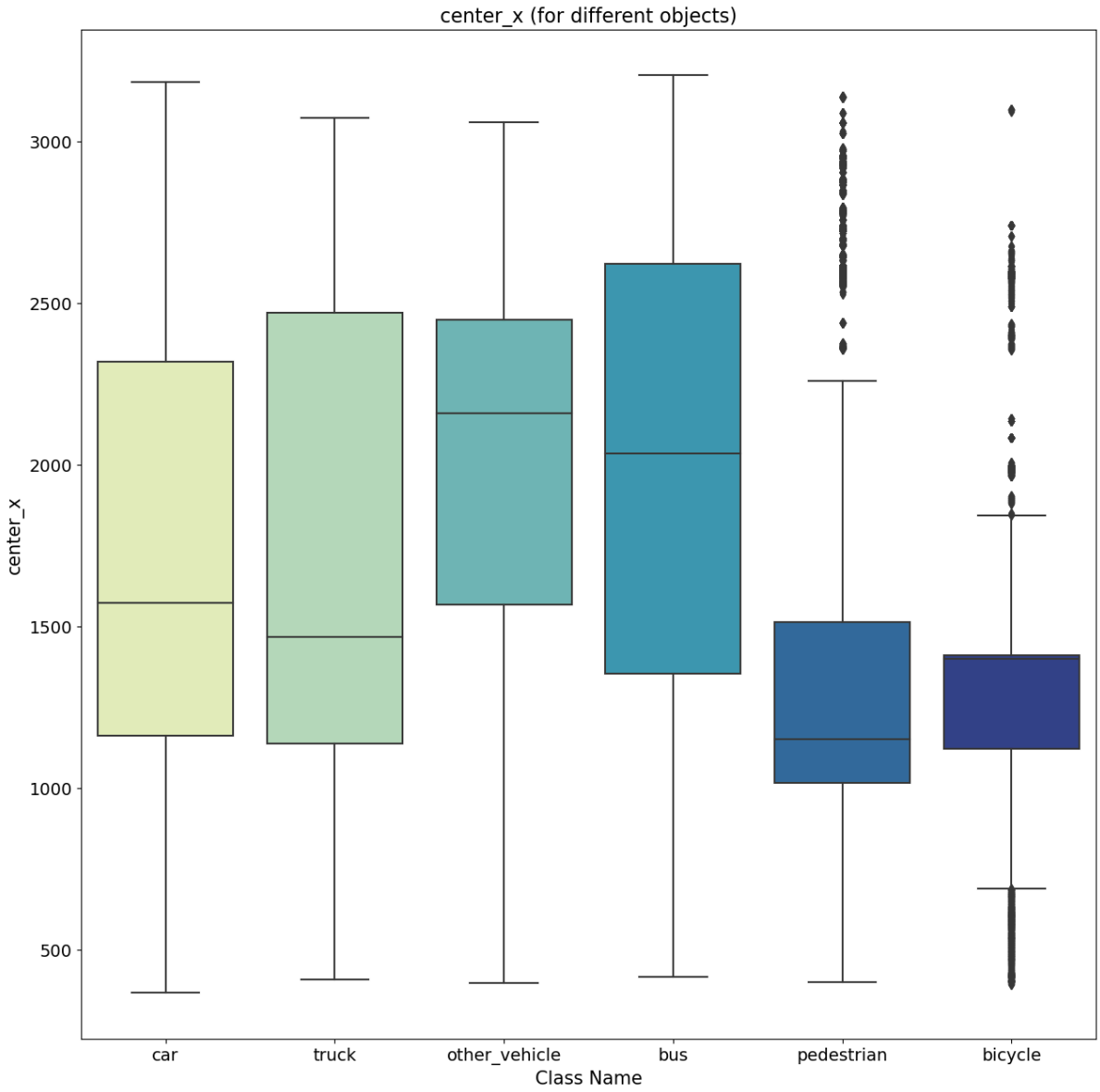


In [22]:

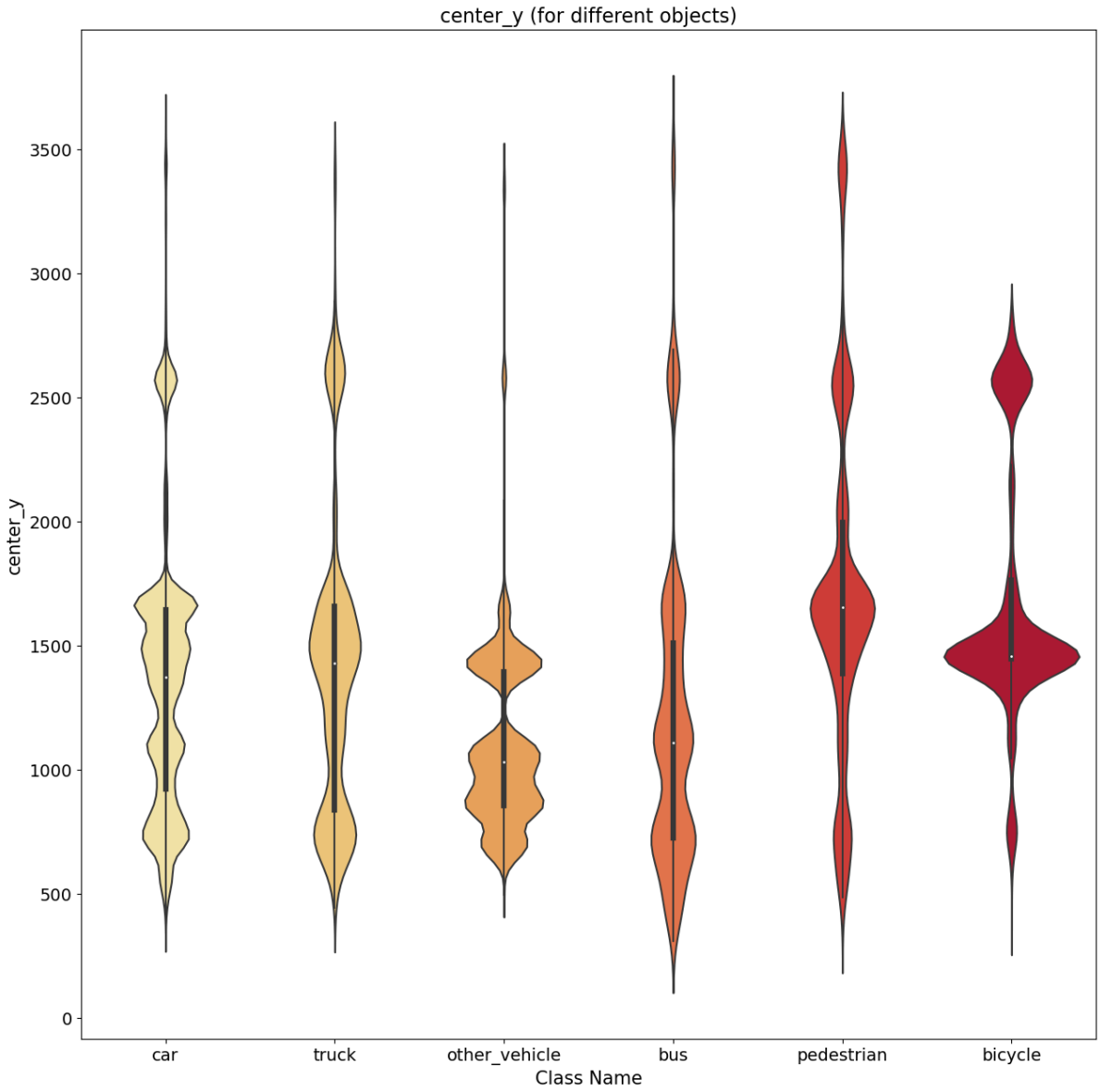
```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.violinplot(x="class_name", y="center_x",
                      data=train_objects.query('class_name != "motorcycle" and clas
                      palette='YlGnBu',
                      split=True, ax=ax).set_title('center_x (for different objects

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_x", fontsize=15)
plt.show(plot)
```



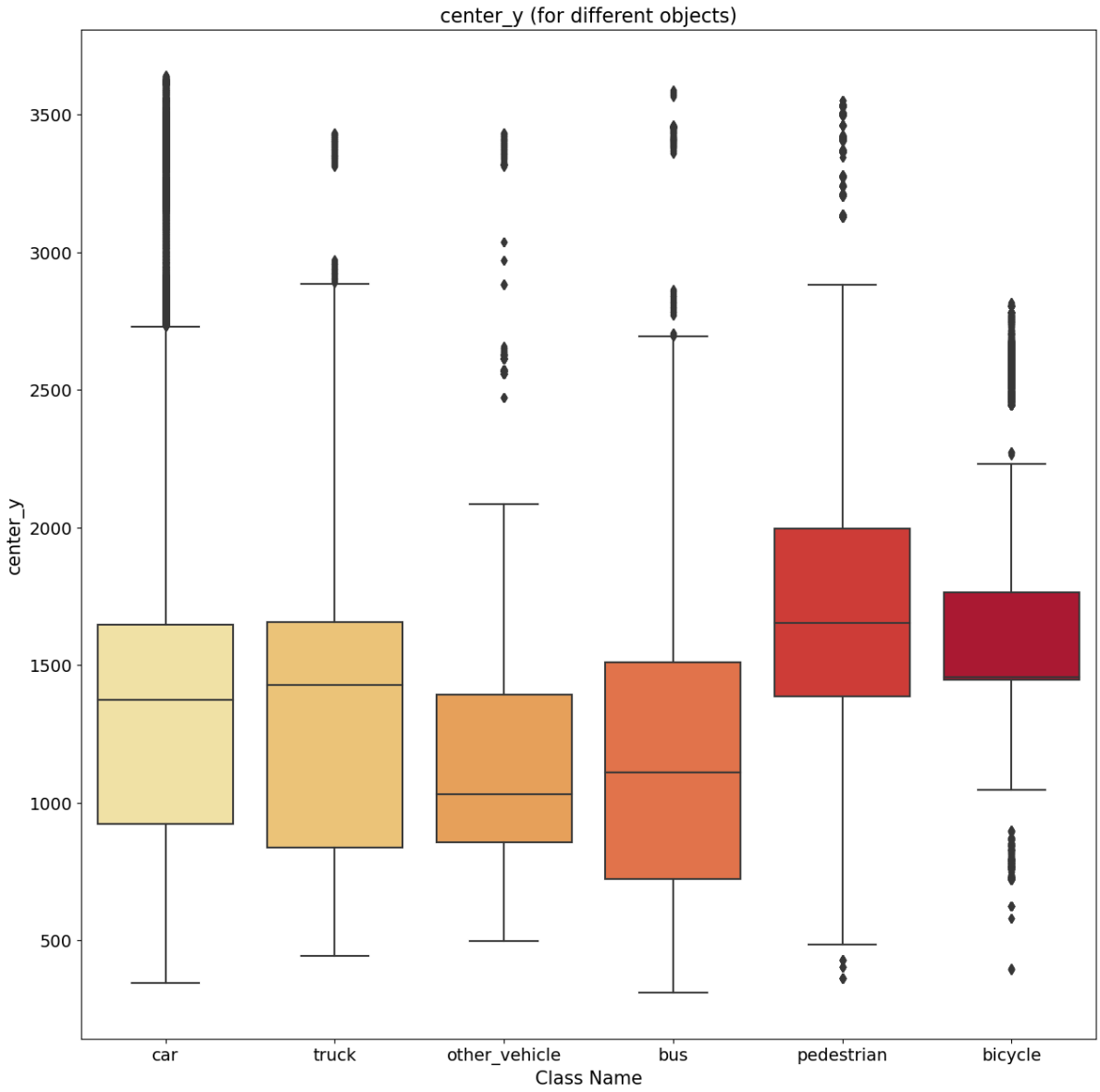
In [23]:
```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.boxplot(x="class_name", y="center_x",
                   data=train_objects.query('class_name != "motorcycle" and class_r
                   palette='YlGnBu', ax=ax).set_title('center_x (for different obje

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_x", fontsize=15)
plt.show(plot)
```



In [24]:
```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.violinplot(x="class_name", y="center_y",
                      data=train_objects.query('class_name != "motorcycle" and clas
                      palette='YlOrRd',
                      split=True, ax=ax).set_title('center_y (for different objects

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_y", fontsize=15)
plt.show(plot)
```

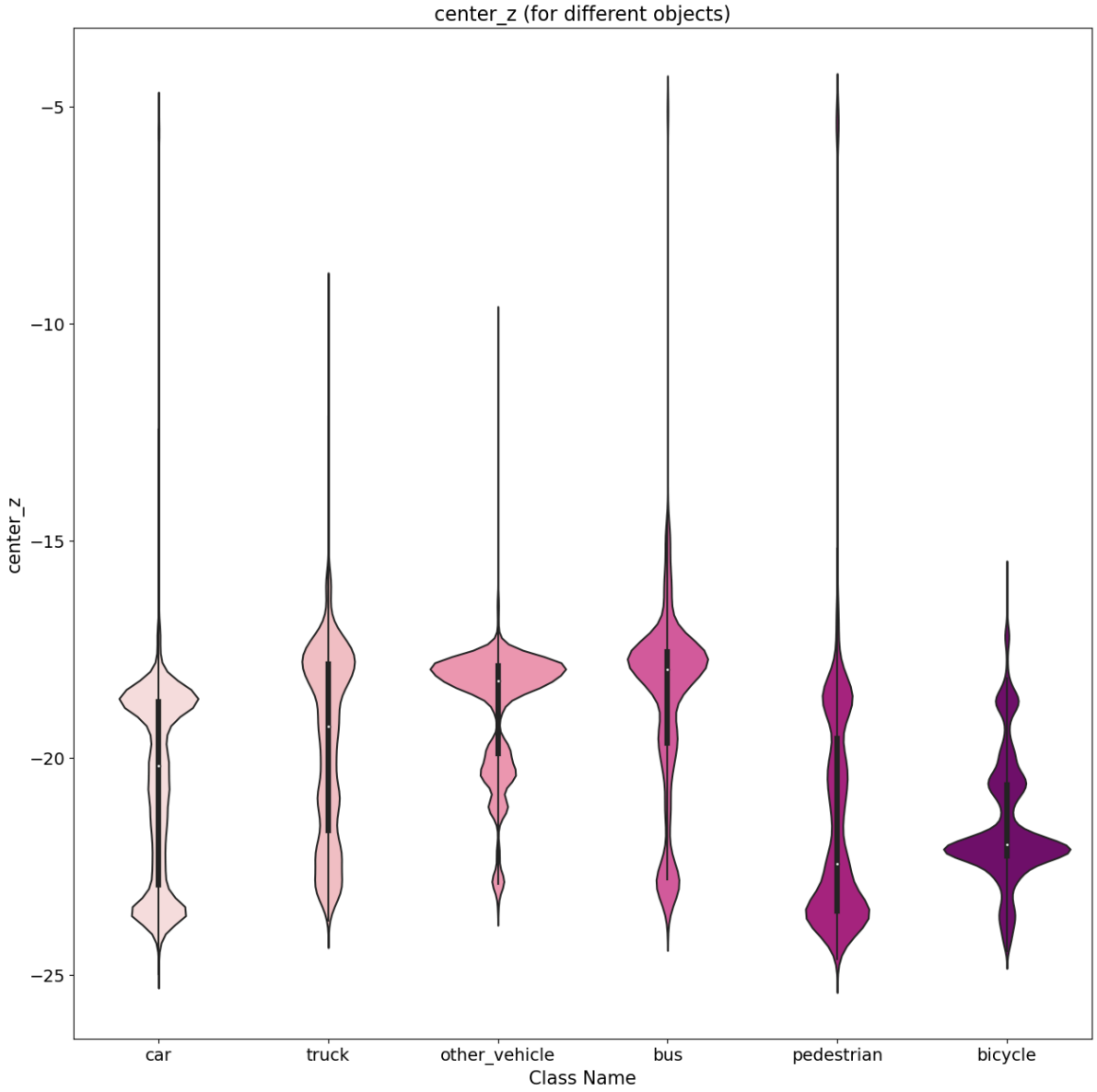


In [25]:
```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.boxplot(x="class_name", y="center_y",
                   data=train_objects.query('class_name != "motorcycle" and class_r
                   palette='YlOrRd', ax=ax).set_title('center_y (for different obje

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_y", fontsize=15)
plt.show(plot)
```

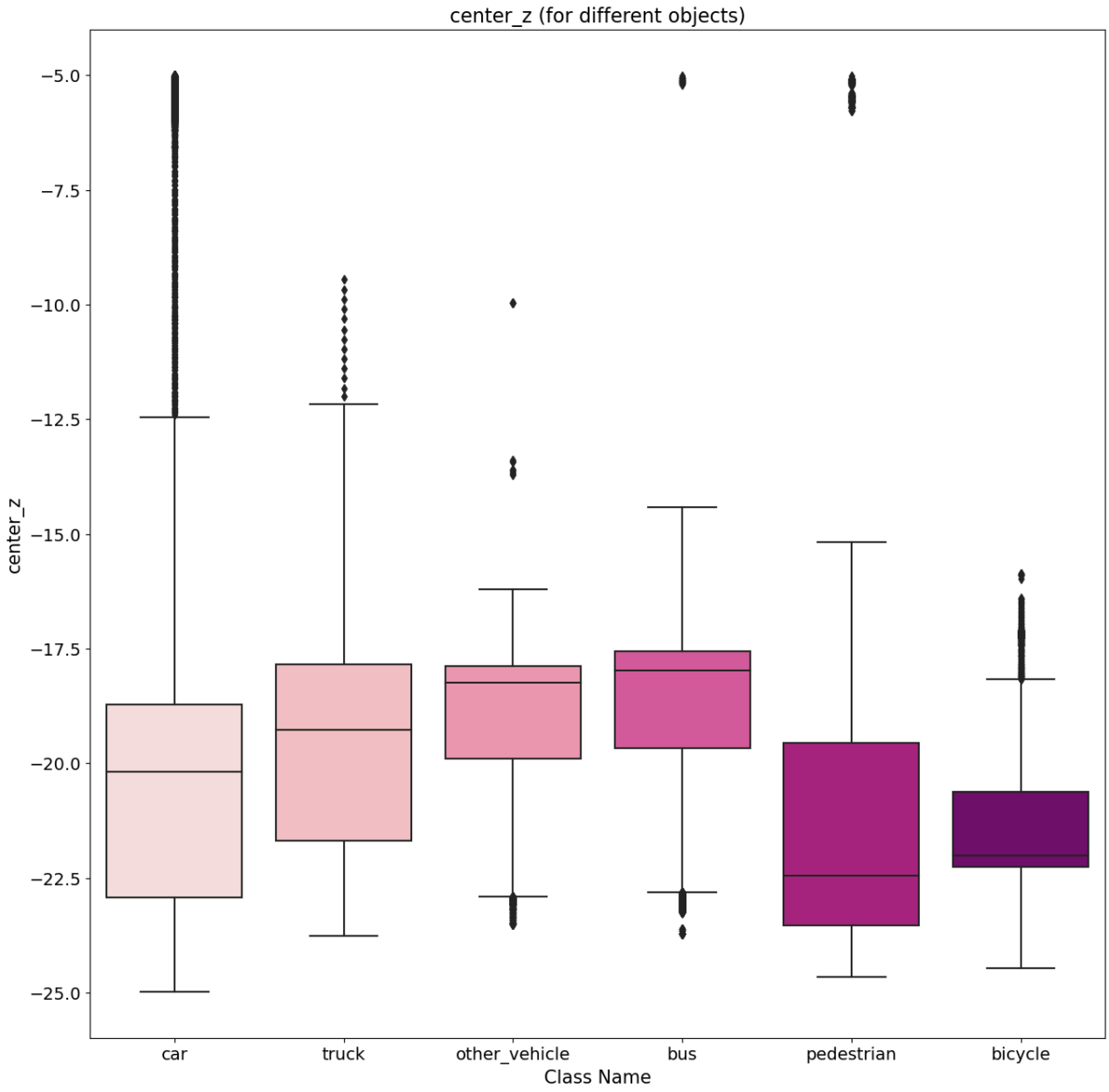


In [26]:

```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.violinplot(x="class_name", y="center_z",
                      data=train_objects.query('class_name != "motorcycle" and clas
                      palette='RdPu',
                      split=True, ax=ax).set_title('center_z (for different objects

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_z", fontsize=15)
plt.show(plot)
```



In [27]:
```
fig, ax = plt.subplots(figsize=(15, 15))

plot = sns.boxplot(x="class_name", y="center_z",
                   data=train_objects.query('class_name != "motorcycle" and class_r
                   palette='RdPu', ax=ax).set_title('center_z (for different object

plt.yticks(fontsize=14)
plt.xticks(fontsize=14)
plt.xlabel("Class Name", fontsize=15)
plt.ylabel("center_z", fontsize=15)
plt.show(plot)
```



In [28]:

```python
class PointCloud(ABC):
    """
    Abstract class for manipulating and viewing point clouds.
    Every point cloud (lidar and radar) consists of points where:
    - Dimensions 0, 1, 2 represent x, y, z coordinates.
        These are modified when the point cloud is rotated or translated.
    - All other dimensions are optional. Hence these have to be manually modified i
    """

    def __init__(self, points: np.ndarray):
        """
        Initialize a point cloud and check it has the correct dimensions.
        :param points: <np.float: d, n>. d-dimensional input point cloud matrix.
        """
        assert points.shape[0] == self.nbr_dims(), (
            "Error: Pointcloud points must have format: %d x n" % self.nbr_dims()
        )
        self.points = points

    @staticmethod
    @abstractmethod
    def nbr_dims() -> int:
        """Returns the number of dimensions.
        Returns: Number of dimensions.
        """
        pass
```

```
    @classmethod
    @abstractmethod
    def from_file(cls, file_name: str) -> "PointCloud":
        """Loads point cloud from disk.
        Args:
            file_name: Path of the pointcloud file on disk.
        Returns: PointCloud instance.
        """
        pass

    @classmethod
    def from_file_multisweep(
        cls, lyftd, sample_rec: Dict, chan: str, ref_chan: str, num_sweeps: int = 2
    ) -> Tuple["PointCloud", np.ndarray]:
        """Return a point cloud that aggregates multiple sweeps.
        As every sweep is in a different coordinate frame, we need to map the coord
        As every sweep has a different timestamp, we need to account for that in th
        Args:
            lyftd: A LyftDataset instance.
            sample_rec: The current sample.
            chan: The radar channel from which we track back n sweeps to aggregate
            ref_chan: The reference channel of the current sample_rec that the poin
            num_sweeps: Number of sweeps to aggregated.
            min_distance: Distance below which points are discarded.
        Returns: (all_pc, all_times). The aggregated point cloud and timestamps.
        """

        # Init
        points = np.zeros((cls.nbr_dims(), 0))
        all_pc = cls(points)
        all_times = np.zeros((1, 0))

        # Get reference pose and timestamp
        ref_sd_token = sample_rec["data"][ref_chan]
        ref_sd_rec = lyftd.get("sample_data", ref_sd_token)
        ref_pose_rec = lyftd.get("ego_pose", ref_sd_rec["ego_pose_token"])
        ref_cs_rec = lyftd.get("calibrated_sensor", ref_sd_rec["calibrated_sensor_t
        ref_time = 1e-6 * ref_sd_rec["timestamp"]

        # Homogeneous transform from ego car frame to reference frame
        ref_from_car = transform_matrix(ref_cs_rec["translation"], Quaternion(ref_c

        # Homogeneous transformation matrix from global to _current_ ego car frame
        car_from_global = transform_matrix(
            ref_pose_rec["translation"], Quaternion(ref_pose_rec["rotation"]), inve
        )

        # Aggregate current and previous sweeps.
        sample_data_token = sample_rec["data"][chan]
        current_sd_rec = lyftd.get("sample_data", sample_data_token)
        for _ in range(num_sweeps):
            # Load up the pointcloud.
            current_pc = cls.from_file(lyftd.data_path / ('train_' + current_sd_rec

            # Get past pose.
            current_pose_rec = lyftd.get("ego_pose", current_sd_rec["ego_pose_token
            global_from_car = transform_matrix(
                current_pose_rec["translation"], Quaternion(current_pose_rec["rotat
            )

            # Homogeneous transformation matrix from sensor coordinate frame to ego
            current_cs_rec = lyftd.get("calibrated_sensor", current_sd_rec["calibra
            car_from_current = transform_matrix(
                current_cs_rec["translation"], Quaternion(current_cs_rec["rotation"
```

```
            )

            # Fuse four transformation matrices into one and perform transform.
            trans_matrix = reduce(np.dot, [ref_from_car, car_from_global, global_fr
            current_pc.transform(trans_matrix)

            # Remove close points and add timevector.
            current_pc.remove_close(min_distance)
            time_lag = ref_time - 1e-6 * current_sd_rec["timestamp"]  # positive di
            times = time_lag * np.ones((1, current_pc.nbr_points()))
            all_times = np.hstack((all_times, times))

            # Merge with key pc.
            all_pc.points = np.hstack((all_pc.points, current_pc.points))

            # Abort if there are no previous sweeps.
            if current_sd_rec["prev"] == "":
                break
            else:
                current_sd_rec = lyftd.get("sample_data", current_sd_rec["prev"])

        return all_pc, all_times

    def nbr_points(self) -> int:
        """Returns the number of points."""
        return self.points.shape[1]

    def subsample(self, ratio: float) -> None:
        """Sub-samples the pointcloud.
        Args:
            ratio: Fraction to keep.
        """
        selected_ind = np.random.choice(np.arange(0, self.nbr_points()), size=int(s
        self.points = self.points[:, selected_ind]

    def remove_close(self, radius: float) -> None:
        """Removes point too close within a certain radius from origin.
        Args:
            radius: Radius below which points are removed.
        Returns:
        """
        x_filt = np.abs(self.points[0, :]) < radius
        y_filt = np.abs(self.points[1, :]) < radius
        not_close = np.logical_not(np.logical_and(x_filt, y_filt))
        self.points = self.points[:, not_close]

    def translate(self, x: np.ndarray) -> None:
        """Applies a translation to the point cloud.
        Args:
            x: <np.float: 3, 1>. Translation in x, y, z.
        """
        for i in range(3):
            self.points[i, :] = self.points[i, :] + x[i]

    def rotate(self, rot_matrix: np.ndarray) -> None:
        """Applies a rotation.
        Args:
            rot_matrix: <np.float: 3, 3>. Rotation matrix.
        Returns:
        """
        self.points[:3, :] = np.dot(rot_matrix, self.points[:3, :])

    def transform(self, transf_matrix: np.ndarray) -> None:
        """Applies a homogeneous transform.
```

```
        Args:
            transf_matrix: transf_matrix: <np.float: 4, 4>. Homogenous transformati
        """
        self.points[:3, :] = transf_matrix.dot(np.vstack((self.points[:3, :], np.or

    def render_height(
        self,
        ax: Axes,
        view: np.ndarray = np.eye(4),
        x_lim: Tuple = (-20, 20),
        y_lim: Tuple = (-20, 20),
        marker_size: float = 1,
    ) -> None:
        """Simple method that applies a transformation and then scatter plots the p
        Args:
            ax: Axes on which to render the points.
            view: <np.float: n, n>. Defines an arbitrary projection (n <= 4).
            x_lim: (min <float>, max <float>). x range for plotting.
            y_lim: (min <float>, max <float>). y range for plotting.
            marker_size: Marker size.
        """
        self._render_helper(2, ax, view, x_lim, y_lim, marker_size)

    def render_intensity(
        self,
        ax: Axes,
        view: np.ndarray = np.eye(4),
        x_lim: Tuple = (-20, 20),
        y_lim: Tuple = (-20, 20),
        marker_size: float = 1,
    ) -> None:
        """Very simple method that applies a transformation and then scatter plots
        Args:
            ax: Axes on which to render the points.
            view: <np.float: n, n>. Defines an arbitrary projection (n <= 4).
            x_lim: (min <float>, max <float>).
            y_lim: (min <float>, max <float>).
            marker_size: Marker size.
        Returns:
        """
        self._render_helper(3, ax, view, x_lim, y_lim, marker_size)

    def _render_helper(
        self, color_channel: int, ax: Axes, view: np.ndarray, x_lim: Tuple, y_lim:
    ) -> None:
        """Helper function for rendering.
        Args:
            color_channel: Point channel to use as color.
            ax: Axes on which to render the points.
            view: <np.float: n, n>. Defines an arbitrary projection (n <= 4).
            x_lim: (min <float>, max <float>).
            y_lim: (min <float>, max <float>).
            marker_size: Marker size.
        """
        points = view_points(self.points[:3, :], view, normalize=False)
        ax.scatter(points[0, :], points[1, :], c=self.points[color_channel, :], s=m
        ax.set_xlim(x_lim[0], x_lim[1])
        ax.set_ylim(y_lim[0], y_lim[1])


class LidarPointCloud(PointCloud):
    @staticmethod
    def nbr_dims() -> int:
        """Returns the number of dimensions.
```

```
        Returns: Number of dimensions.
        """
        return 4

    @classmethod
    def from_file(cls, file_name: Path) -> "LidarPointCloud":
        """Loads LIDAR data from binary numpy format. Data is stored as (x, y, z, i
        Args:
            file_name: Path of the pointcloud file on disk.
        Returns: LidarPointCloud instance (x, y, z, intensity).
        """

        assert file_name.suffix == ".bin", "Unsupported filetype {}".format(file_na

        scan = np.fromfile(str(file_name), dtype=np.float32)
        points = scan.reshape((-1, 5))[:, : cls.nbr_dims()]
        return cls(points.T)


class RadarPointCloud(PointCloud):

    # Class-level settings for radar pointclouds, see from_file().
    invalid_states = [0]  # type: List[int]
    dynprop_states = range(7)  # type: List[int] # Use [0, 2, 6] for moving objects
    ambig_states = [3]  # type: List[int]

    @staticmethod
    def nbr_dims() -> int:
        """Returns the number of dimensions.
        Returns: Number of dimensions.
        """
        return 18

    @classmethod
    def from_file(
        cls,
        file_name: Path,
        invalid_states: List[int] = None,
        dynprop_states: List[int] = None,
        ambig_states: List[int] = None,
    ) -> "RadarPointCloud":
        """Loads RADAR data from a Point Cloud Data file. See details below.
        Args:
            file_name: The path of the pointcloud file.
            invalid_states: Radar states to be kept. See details below.
            dynprop_states: Radar states to be kept. Use [0, 2, 6] for moving objec
            ambig_states: Radar states to be kept. See details below. To keep all r
                set each state filter to range(18).
        Returns: <np.float: d, n>. Point cloud matrix with d dimensions and n point
        Example of the header fields:
        # .PCD v0.7 - Point Cloud Data file format
        VERSION 0.7
        FIELDS x y z dyn_prop id rcs vx vy vx_comp vy_comp is_quality_valid ambig_
                                                              state x_rms y_rms inval
        SIZE 4 4 4 1 2 4 4 4 4 4 1 1 1 1 1 1 1
        TYPE F F F I I F F F F F I I I I I I I
        COUNT 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
        WIDTH 125
        HEIGHT 1
        VIEWPOINT 0 0 0 1 0 0 0
        POINTS 125
        DATA binary
        Below some of the fields are explained in more detail:
        x is front, y is left
```

```
        vx, vy are the velocities in m/s.
        vx_comp, vy_comp are the velocities in m/s compensated by the ego motion.
        We recommend using the compensated velocities.
        invalid_state: state of Cluster validity state.
        (Invalid states)
        0x01    invalid due to low RCS
        0x02    invalid due to near-field artefact
        0x03    invalid far range cluster because not confirmed in near range
        0x05    reserved
        0x06    invalid cluster due to high mirror probability
        0x07    Invalid cluster because outside sensor field of view
        0x0d    reserved
        0x0e    invalid cluster because it is a harmonics
        (Valid states)
        0x00    valid
        0x04    valid cluster with low RCS
        0x08    valid cluster with azimuth correction due to elevation
        0x09    valid cluster with high child probability
        0x0a    valid cluster with high probability of being a 50 deg artefact
        0x0b    valid cluster but no local maximum
        0x0c    valid cluster with high artefact probability
        0x0f    valid cluster with above 95m in near range
        0x10    valid cluster with high multi-target probability
        0x11    valid cluster with suspicious angle
        dynProp: Dynamic property of cluster to indicate if is moving or not.
        0: moving
        1: stationary
        2: oncoming
        3: stationary candidate
        4: unknown
        5: crossing stationary
        6: crossing moving
        7: stopped
        ambig_state: State of Doppler (radial velocity) ambiguity solution.
        0: invalid
        1: ambiguous
        2: staggered ramp
        3: unambiguous
        4: stationary candidates
        pdh0: False alarm probability of cluster (i.e. probability of being an arte

        0: invalid
        1: <25%
        2: 50%
        3: 75%
        4: 90%
        5: 99%
        6: 99.9%
        7: <=100%
        """

        assert file_name.suffix == ".pcd", "Unsupported filetype {}".format(file_na

        meta = []
        with open(str(file_name), "rb") as f:
            for line in f:
                line = line.strip().decode("utf-8")
                meta.append(line)
                if line.startswith("DATA"):
                    break

            data_binary = f.read()

        # Get the header rows and check if they appear as expected.
```

```
        assert meta[0].startswith("#"), "First line must be comment"
        assert meta[1].startswith("VERSION"), "Second line must be VERSION"
        sizes = meta[3].split(" ")[1:]
        types = meta[4].split(" ")[1:]
        counts = meta[5].split(" ")[1:]
        width = int(meta[6].split(" ")[1])
        height = int(meta[7].split(" ")[1])
        data = meta[10].split(" ")[1]
        feature_count = len(types)
        assert width > 0
        assert len([c for c in counts if c != c]) == 0, "Error: COUNT not supported
        assert height == 1, "Error: height != 0 not supported!"
        assert data == "binary"

        # Lookup table for how to decode the binaries.
        unpacking_lut = {
            "F": {2: "e", 4: "f", 8: "d"},
            "I": {1: "b", 2: "h", 4: "i", 8: "q"},
            "U": {1: "B", 2: "H", 4: "I", 8: "Q"},
        }
        types_str = "".join([unpacking_lut[t][int(s)] for t, s in zip(types, sizes)

        # Decode each point.
        offset = 0
        point_count = width
        points = []
        for i in range(point_count):
            point = []
            for p in range(feature_count):
                start_p = offset
                end_p = start_p + int(sizes[p])
                assert end_p < len(data_binary)
                point_p = struct.unpack(types_str[p], data_binary[start_p:end_p])[0
                point.append(point_p)
                offset = end_p
            points.append(point)

        # A NaN in the first point indicates an empty pointcloud.
        point = np.array(points[0])
        if np.any(np.isnan(point)):
            return cls(np.zeros((feature_count, 0)))

        # Convert to numpy matrix.
        points = np.array(points).transpose()

        # If no parameters are provided, use default settings.
        invalid_states = cls.invalid_states if invalid_states is None else invalid_
        dynprop_states = cls.dynprop_states if dynprop_states is None else dynprop_
        ambig_states = cls.ambig_states if ambig_states is None else ambig_states

        # Filter points with an invalid state.
        valid = [p in invalid_states for p in points[-4, :]]
        points = points[:, valid]

        # Filter by dynProp.
        valid = [p in dynprop_states for p in points[3, :]]
        points = points[:, valid]

        # Filter by ambig_state.
        valid = [p in ambig_states for p in points[11, :]]
        points = points[:, valid]

        return cls(points)
```

```
class Box:
    """ Simple data class representing a 3d box including, label, score and velocit

    def __init__(
        self,
        center: List[float],
        size: List[float],
        orientation: Quaternion,
        label: int = np.nan,
        score: float = np.nan,
        velocity: Tuple = (np.nan, np.nan, np.nan),
        name: str = None,
        token: str = None,
    ):
        """
        Args:
            center: Center of box given as x, y, z.
            size: Size of box in width, length, height.
            orientation: Box orientation.
            label: Integer label, optional.
            score: Classification score, optional.
            velocity: Box velocity in x, y, z direction.
            name: Box name, optional. Can be used e.g. for denote category name.
            token: Unique string identifier from DB.
        """
        assert not np.any(np.isnan(center))
        assert not np.any(np.isnan(size))
        assert len(center) == 3
        assert len(size) == 3
        assert type(orientation) == Quaternion

        self.center = np.array(center)
        self.wlh = np.array(size)
        self.orientation = orientation
        self.label = int(label) if not np.isnan(label) else label
        self.score = float(score) if not np.isnan(score) else score
        self.velocity = np.array(velocity)
        self.name = name
        self.token = token

    def __eq__(self, other):
        center = np.allclose(self.center, other.center)
        wlh = np.allclose(self.wlh, other.wlh)
        orientation = np.allclose(self.orientation.elements, other.orientation.elem
        label = (self.label == other.label) or (np.isnan(self.label) and np.isnan(o
        score = (self.score == other.score) or (np.isnan(self.score) and np.isnan(o
        vel = np.allclose(self.velocity, other.velocity) or (
            np.all(np.isnan(self.velocity)) and np.all(np.isnan(other.velocity))
        )

        return center and wlh and orientation and label and score and vel

    def __repr__(self):
        repr_str = (
            "label: {}, score: {:.2f}, xyz: [{:.2f}, {:.2f}, {:.2f}], wlh: [{:.2f},
            "rot axis: [{:.2f}, {:.2f}, {:.2f}], ang(degrees): {:.2f}, ang(rad): {:
            "vel: {:.2f}, {:.2f}, {:.2f}, name: {}, token: {}"
        )

        return repr_str.format(
            self.label,
            self.score,
            self.center[0],
```

```
            self.center[1],
            self.center[2],
            self.wlh[0],
            self.wlh[1],
            self.wlh[2],
            self.orientation.axis[0],
            self.orientation.axis[1],
            self.orientation.axis[2],
            self.orientation.degrees,
            self.orientation.radians,
            self.velocity[0],
            self.velocity[1],
            self.velocity[2],
            self.name,
            self.token,
        )

    @property
    def rotation_matrix(self) -> np.ndarray:
        """Return a rotation matrix.
        Returns: <np.float: 3, 3>. The box's rotation matrix.
        """
        return self.orientation.rotation_matrix

    def translate(self, x: np.ndarray) -> None:
        """Applies a translation.
        Args:
            x: <np.float: 3, 1>. Translation in x, y, z direction.
        """
        self.center += x

    def rotate(self, quaternion: Quaternion) -> None:
        """Rotates box.
        Args:
            quaternion: Rotation to apply.
        """
        self.center = np.dot(quaternion.rotation_matrix, self.center)
        self.orientation = quaternion * self.orientation
        self.velocity = np.dot(quaternion.rotation_matrix, self.velocity)

    def corners(self, wlh_factor: float = 1.0) -> np.ndarray:
        """Returns the bounding box corners.
        Args:
            wlh_factor: Multiply width, length, height by a factor to scale the box
        Returns: First four corners are the ones facing forward.
                 The last four are the ones facing backwards.
        """

        width, length, height = self.wlh * wlh_factor

        # 3D bounding box corners. (Convention: x points forward, y to the left, z
        x_corners = length / 2 * np.array([1, 1, 1, 1, -1, -1, -1, -1])
        y_corners = width / 2 * np.array([1, -1, -1, 1, 1, -1, -1, 1])
        z_corners = height / 2 * np.array([1, 1, -1, -1, 1, 1, -1, -1])
        corners = np.vstack((x_corners, y_corners, z_corners))

        # Rotate
        corners = np.dot(self.orientation.rotation_matrix, corners)

        # Translate
        x, y, z = self.center
        corners[0, :] = corners[0, :] + x
        corners[1, :] = corners[1, :] + y
        corners[2, :] = corners[2, :] + z
```

```
        return corners

    def bottom_corners(self) -> np.ndarray:
        """Returns the four bottom corners.
        Returns: <np.float: 3, 4>. Bottom corners. First two face forward, last two
        """
        return self.corners()[:, [2, 3, 7, 6]]

    def render(
        self,
        axis: Axes,
        view: np.ndarray = np.eye(3),
        normalize: bool = False,
        colors: Tuple = ("b", "r", "k"),
        linewidth: float = 2,
    ):
        """Renders the box in the provided Matplotlib axis.
        Args:
            axis: Axis onto which the box should be drawn.
            view: <np.array: 3, 3>. Define a projection in needed (e.g. for drawing
            normalize: Whether to normalize the remaining coordinate.
            colors: (<Matplotlib.colors>: 3). Valid Matplotlib colors (<str> or nor
            back and sides.
            linewidth: Width in pixel of the box sides.
        """
        corners = view_points(self.corners(), view, normalize=normalize)[:2, :]

        def draw_rect(selected_corners, color):
            prev = selected_corners[-1]
            for corner in selected_corners:
                axis.plot([prev[0], corner[0]], [prev[1], corner[1]], color=color,
                prev = corner

        # Draw the sides
        for i in range(4):
            axis.plot(
                [corners.T[i][0], corners.T[i + 4][0]],
                [corners.T[i][1], corners.T[i + 4][1]],
                color=colors[2],
                linewidth=linewidth,
            )

        # Draw front (first 4 corners) and rear (last 4 corners) rectangles(3d)/lin
        draw_rect(corners.T[:4], colors[0])
        draw_rect(corners.T[4:], colors[1])

        # Draw line indicating the front
        center_bottom_forward = np.mean(corners.T[2:4], axis=0)
        center_bottom = np.mean(corners.T[[2, 3, 7, 6]], axis=0)
        axis.plot(
            [center_bottom[0], center_bottom_forward[0]],
            [center_bottom[1], center_bottom_forward[1]],
            color=colors[0],
            linewidth=linewidth,
        )

    def render_cv2(
        self,
        image: np.ndarray,
        view: np.ndarray = np.eye(3),
        normalize: bool = False,
        colors: Tuple = ((0, 0, 255), (255, 0, 0), (155, 155, 155)),
        linewidth: int = 2,
```

```
    ) -> None:
        """Renders box using OpenCV2.
        Args:
            image: <np.array: width, height, 3>. Image array. Channels are in BGR c
            view: <np.array: 3, 3>. Define a projection if needed (e.g. for drawing
            normalize: Whether to normalize the remaining coordinate.
            colors: ((R, G, B), (R, G, B), (R, G, B)). Colors for front, side & rea
            linewidth: Linewidth for plot.
        Returns:
        """
        corners = view_points(self.corners(), view, normalize=normalize)[:2, :]

        def draw_rect(selected_corners, color):
            prev = selected_corners[-1]
            for corner in selected_corners:
                cv2.line(image, (int(prev[0]), int(prev[1])), (int(corner[0]), int(
                prev = corner

        # Draw the sides
        for i in range(4):
            cv2.line(
                image,
                (int(corners.T[i][0]), int(corners.T[i][1])),
                (int(corners.T[i + 4][0]), int(corners.T[i + 4][1])),
                colors[2][::-1],
                linewidth,
            )

        # Draw front (first 4 corners) and rear (last 4 corners) rectangles(3d)/lin
        draw_rect(corners.T[:4], colors[0][::-1])
        draw_rect(corners.T[4:], colors[1][::-1])

        # Draw line indicating the front
        center_bottom_forward = np.mean(corners.T[2:4], axis=0)
        center_bottom = np.mean(corners.T[[2, 3, 7, 6]], axis=0)
        cv2.line(
            image,
            (int(center_bottom[0]), int(center_bottom[1])),
            (int(center_bottom_forward[0]), int(center_bottom_forward[1])),
            colors[0][::-1],
            linewidth,
        )

    def copy(self) -> "Box":
        """        Create a copy of self.
        Returns: A copy.
        """
        return copy.deepcopy(self)
```

In [29]:

```
PYTHON_VERSION = sys.version_info[0]

if not PYTHON_VERSION == 3:
    raise ValueError("LyftDataset sdk only supports Python version 3.")


class LyftDataset:
    """Database class for Lyft Dataset to help query and retrieve information from

    def __init__(self, data_path: str, json_path: str, verbose: bool = True, map_re
        """Loads database and creates reverse indexes and shortcuts.
        Args:
            data_path: Path to the tables and data.
            json_path: Path to the folder with json files
            verbose: Whether to print status messages during load.
```

```
            map_resolution: Resolution of maps (meters).
        """

        self.data_path = Path(data_path).expanduser().absolute()
        self.json_path = Path(json_path)

        self.table_names = [
            "category",
            "attribute",
            "visibility",
            "instance",
            "sensor",
            "calibrated_sensor",
            "ego_pose",
            "log",
            "scene",
            "sample",
            "sample_data",
            "sample_annotation",
            "map",
        ]

        start_time = time.time()

        # Explicitly assign tables to help the IDE determine valid class members.
        self.category = self.__load_table__("category")
        self.attribute = self.__load_table__("attribute")
        self.visibility = self.__load_table__("visibility")
        self.instance = self.__load_table__("instance")
        self.sensor = self.__load_table__("sensor")
        self.calibrated_sensor = self.__load_table__("calibrated_sensor")
        self.ego_pose = self.__load_table__("ego_pose")
        self.log = self.__load_table__("log")
        self.scene = self.__load_table__("scene")
        self.sample = self.__load_table__("sample")
        self.sample_data = self.__load_table__("sample_data")
        self.sample_annotation = self.__load_table__("sample_annotation")
        self.map = self.__load_table__("map")

        # Initialize map mask for each map record.
        for map_record in self.map:
            map_record["mask"] = MapMask(self.data_path / 'train_maps/map_raster_pa

        if verbose:
            for table in self.table_names:
                print("{} {},".format(len(getattr(self, table)), table))
            print("Done loading in {:.1f} seconds.\n======".format(time.time() - st

        # Make reverse indexes for common lookups.
        self.__make_reverse_index__(verbose)

        # Initialize LyftDatasetExplorer class
        self.explorer = LyftDatasetExplorer(self)

    def __load_table__(self, table_name) -> dict:
        """Loads a table."""
        with open(str(self.json_path.joinpath("{}.json".format(table_name)))) as f:
            table = json.load(f)
        return table

    def __make_reverse_index__(self, verbose: bool) -> None:
        """De-normalizes database to create reverse indices for common cases.
        Args:
            verbose: Whether to print outputs.
```

```
        """

        start_time = time.time()
        if verbose:
            print("Reverse indexing ...")

        # Store the mapping from token to table index for each table.
        self._token2ind = dict()
        for table in self.table_names:
            self._token2ind[table] = dict()

            for ind, member in enumerate(getattr(self, table)):
                self._token2ind[table][member["token"]] = ind

        # Decorate (adds short-cut) sample_annotation table with for category name.
        for record in self.sample_annotation:
            inst = self.get("instance", record["instance_token"])
            record["category_name"] = self.get("category", inst["category_token"])[

        # Decorate (adds short-cut) sample_data with sensor information.
        for record in self.sample_data:
            cs_record = self.get("calibrated_sensor", record["calibrated_sensor_tok
            sensor_record = self.get("sensor", cs_record["sensor_token"])
            record["sensor_modality"] = sensor_record["modality"]
            record["channel"] = sensor_record["channel"]

        # Reverse-index samples with sample_data and annotations.
        for record in self.sample:
            record["data"] = {}
            record["anns"] = []

        for record in self.sample_data:
            if record["is_key_frame"]:
                sample_record = self.get("sample", record["sample_token"])
                sample_record["data"][record["channel"]] = record["token"]

        for ann_record in self.sample_annotation:
            sample_record = self.get("sample", ann_record["sample_token"])
            sample_record["anns"].append(ann_record["token"])

        # Add reverse indices from log records to map records.
        if "log_tokens" not in self.map[0].keys():
            raise Exception("Error: log_tokens not in map table. This code is not c
        log_to_map = dict()
        for map_record in self.map:
            for log_token in map_record["log_tokens"]:
                log_to_map[log_token] = map_record["token"]
        for log_record in self.log:
            log_record["map_token"] = log_to_map[log_record["token"]]

        if verbose:
            print("Done reverse indexing in {:.1f} seconds.\n======".format(time.ti

    def get(self, table_name: str, token: str) -> dict:
        """Returns a record from table in constant runtime.
        Args:
            table_name: Table name.
            token: Token of the record.
        Returns: Table record.
        """

        assert table_name in self.table_names, "Table {} not found".format(table_na

        return getattr(self, table_name)[self.getind(table_name, token)]
```

```
    def getind(self, table_name: str, token: str) -> int:
        """Returns the index of the record in a table in constant runtime.
        Args:
            table_name: Table name.
            token: The index of the record in table, table is an array.
        Returns:
        """
        return self._token2ind[table_name][token]

    def field2token(self, table_name: str, field: str, query) -> List[str]:
        """Query all records for a certain field value, and returns the tokens for
        Runs in linear time.
        Args:
            table_name: Table name.
            field: Field name.
            query: Query to match against. Needs to type match the content of the q
        Returns: List of tokens for the matching records.
        """
        matches = []
        for member in getattr(self, table_name):
            if member[field] == query:
                matches.append(member["token"])
        return matches

    def get_sample_data_path(self, sample_data_token: str) -> Path:
        """Returns the path to a sample_data.
        Args:
            sample_data_token:
        Returns:
        """

        sd_record = self.get("sample_data", sample_data_token)
        return self.data_path / sd_record["filename"]

    def get_sample_data(
        self,
        sample_data_token: str,
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        selected_anntokens: List[str] = None,
        flat_vehicle_coordinates: bool = False,
    ) -> Tuple[Path, List[Box], np.array]:
        """Returns the data path as well as all annotations related to that sample_
        The boxes are transformed into the current sensor's coordinate frame.
        Args:
            sample_data_token: Sample_data token.
            box_vis_level: If sample_data is an image, this sets required visibilit
            selected_anntokens: If provided only return the selected annotation.
            flat_vehicle_coordinates: Instead of current sensor's coordinate frame,
        aligned to z-plane in world
        Returns: (data_path, boxes, camera_intrinsic <np.array: 3, 3>)
        """

        # Retrieve sensor & pose records
        sd_record = self.get("sample_data", sample_data_token)
        cs_record = self.get("calibrated_sensor", sd_record["calibrated_sensor_toke
        sensor_record = self.get("sensor", cs_record["sensor_token"])
        pose_record = self.get("ego_pose", sd_record["ego_pose_token"])

        data_path = self.get_sample_data_path(sample_data_token)

        if sensor_record["modality"] == "camera":
            cam_intrinsic = np.array(cs_record["camera_intrinsic"])
            imsize = (sd_record["width"], sd_record["height"])
```

```
        else:
            cam_intrinsic = None
            imsize = None

        # Retrieve all sample annotations and map to sensor coordinate system.
        if selected_anntokens is not None:
            boxes = list(map(self.get_box, selected_anntokens))
        else:
            boxes = self.get_boxes(sample_data_token)

        # Make list of Box objects including coord system transforms.
        box_list = []
        for box in boxes:
            if flat_vehicle_coordinates:
                # Move box to ego vehicle coord system parallel to world z plane
                ypr = Quaternion(pose_record["rotation"]).yaw_pitch_roll
                yaw = ypr[0]

                box.translate(-np.array(pose_record["translation"]))
                box.rotate(Quaternion(scalar=np.cos(yaw / 2), vector=[0, 0, np.sin(

            else:
                # Move box to ego vehicle coord system
                box.translate(-np.array(pose_record["translation"]))
                box.rotate(Quaternion(pose_record["rotation"]).inverse)

                #  Move box to sensor coord system
                box.translate(-np.array(cs_record["translation"]))
                box.rotate(Quaternion(cs_record["rotation"]).inverse)

            if sensor_record["modality"] == "camera" and not box_in_image(
                box, cam_intrinsic, imsize, vis_level=box_vis_level
            ):
                continue

            box_list.append(box)

        return data_path, box_list, cam_intrinsic

    def get_box(self, sample_annotation_token: str) -> Box:
        """Instantiates a Box class from a sample annotation record.
        Args:
            sample_annotation_token: Unique sample_annotation identifier.
        Returns:
        """
        record = self.get("sample_annotation", sample_annotation_token)
        return Box(
            record["translation"],
            record["size"],
            Quaternion(record["rotation"]),
            name=record["category_name"],
            token=record["token"],
        )

    def get_boxes(self, sample_data_token: str) -> List[Box]:
        """Instantiates Boxes for all annotation for a particular sample_data recor
        keyframe, this returns the annotations for that sample. But if the sample_d
        sample_data, a linear interpolation is applied to estimate the location of
        sample_data was captured.
        Args:
            sample_data_token: Unique sample_data identifier.
        Returns:
        """
```

```
        # Retrieve sensor & pose records
        sd_record = self.get("sample_data", sample_data_token)
        curr_sample_record = self.get("sample", sd_record["sample_token"])

        if curr_sample_record["prev"] == "" or sd_record["is_key_frame"]:
            # If no previous annotations available, or if sample_data is keyframe j
            boxes = list(map(self.get_box, curr_sample_record["anns"]))

        else:
            prev_sample_record = self.get("sample", curr_sample_record["prev"])

            curr_ann_recs = [self.get("sample_annotation", token) for token in curr
            prev_ann_recs = [self.get("sample_annotation", token) for token in prev

            # Maps instance tokens to prev_ann records
            prev_inst_map = {entry["instance_token"]: entry for entry in prev_ann_r

            t0 = prev_sample_record["timestamp"]
            t1 = curr_sample_record["timestamp"]
            t = sd_record["timestamp"]

            # There are rare situations where the timestamps in the DB are off so e
            t = max(t0, min(t1, t))

            boxes = []
            for curr_ann_rec in curr_ann_recs:

                if curr_ann_rec["instance_token"] in prev_inst_map:
                    # If the annotated instance existed in the previous frame, inte
                    prev_ann_rec = prev_inst_map[curr_ann_rec["instance_token"]]

                    # Interpolate center.
                    center = [
                        np.interp(t, [t0, t1], [c0, c1])
                        for c0, c1 in zip(prev_ann_rec["translation"], curr_ann_rec
                    ]

                    # Interpolate orientation.
                    rotation = Quaternion.slerp(
                        q0=Quaternion(prev_ann_rec["rotation"]),
                        q1=Quaternion(curr_ann_rec["rotation"]),
                        amount=(t - t0) / (t1 - t0),
                    )

                    box = Box(
                        center,
                        curr_ann_rec["size"],
                        rotation,
                        name=curr_ann_rec["category_name"],
                        token=curr_ann_rec["token"],
                    )
                else:
                    # If not, simply grab the current annotation.
                    box = self.get_box(curr_ann_rec["token"])

                boxes.append(box)
        return boxes

    def box_velocity(self, sample_annotation_token: str, max_time_diff: float = 1.5
        """Estimate the velocity for an annotation.
        If possible, we compute the centered difference between the previous and ne
        Otherwise we use the difference between the current and previous/next frame
        If the velocity cannot be estimated, values are set to np.nan.
        Args:
```

```
            sample_annotation_token: Unique sample_annotation identifier.
            max_time_diff: Max allowed time diff between consecutive samples that a
        Returns: <np.float: 3>. Velocity in x/y/z direction in m/s.
        """

        current = self.get("sample_annotation", sample_annotation_token)
        has_prev = current["prev"] != ""
        has_next = current["next"] != ""

        # Cannot estimate velocity for a single annotation.
        if not has_prev and not has_next:
            return np.array([np.nan, np.nan, np.nan])

        if has_prev:
            first = self.get("sample_annotation", current["prev"])
        else:
            first = current

        if has_next:
            last = self.get("sample_annotation", current["next"])
        else:
            last = current

        pos_last = np.array(last["translation"])
        pos_first = np.array(first["translation"])
        pos_diff = pos_last - pos_first

        time_last = 1e-6 * self.get("sample", last["sample_token"])["timestamp"]
        time_first = 1e-6 * self.get("sample", first["sample_token"])["timestamp"]
        time_diff = time_last - time_first

        if has_next and has_prev:
            # If doing centered difference, allow for up to double the max_time_dif
            max_time_diff *= 2

        if time_diff > max_time_diff:
            # If time_diff is too big, don't return an estimate.
            return np.array([np.nan, np.nan, np.nan])
        else:
            return pos_diff / time_diff

    def list_categories(self) -> None:
        self.explorer.list_categories()

    def list_attributes(self) -> None:
        self.explorer.list_attributes()

    def list_scenes(self) -> None:
        self.explorer.list_scenes()

    def list_sample(self, sample_token: str) -> None:
        self.explorer.list_sample(sample_token)

    def render_pointcloud_in_image(
        self,
        sample_token: str,
        dot_size: int = 5,
        pointsensor_channel: str = "LIDAR_TOP",
        camera_channel: str = "CAM_FRONT",
        out_path: str = None,
    ) -> None:
        self.explorer.render_pointcloud_in_image(
            sample_token,
            dot_size,
```

```
            pointsensor_channel=pointsensor_channel,
            camera_channel=camera_channel,
            out_path=out_path,
        )

    def render_sample(
        self,
        sample_token: str,
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        nsweeps: int = 1,
        out_path: str = None,
    ) -> None:
        self.explorer.render_sample(sample_token, box_vis_level, nsweeps=nsweeps, c

    def render_sample_data(
        self,
        sample_data_token: str,
        with_anns: bool = True,
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        axes_limit: float = 40,
        ax: Axes = None,
        nsweeps: int = 1,
        out_path: str = None,
        underlay_map: bool = False,
    ) -> None:
        return self.explorer.render_sample_data(
            sample_data_token,
            with_anns,
            box_vis_level,
            axes_limit,
            ax,
            num_sweeps=nsweeps,
            out_path=out_path,
            underlay_map=underlay_map,
        )

    def render_annotation(
        self,
        sample_annotation_token: str,
        margin: float = 10,
        view: np.ndarray = np.eye(4),
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        out_path: str = None,
    ) -> None:
        self.explorer.render_annotation(sample_annotation_token, margin, view, box_

    def render_instance(self, instance_token: str, out_path: str = None) -> None:
        self.explorer.render_instance(instance_token, out_path=out_path)

    def render_scene(self, scene_token: str, freq: float = 10, imwidth: int = 640,
        self.explorer.render_scene(scene_token, freq, image_width=imwidth, out_path

    def render_scene_channel(
        self,
        scene_token: str,
        channel: str = "CAM_FRONT",
        freq: float = 10,
        imsize: Tuple[float, float] = (640, 360),
        out_path: str = None,
    ) -> None:
        self.explorer.render_scene_channel(
            scene_token=scene_token, channel=channel, freq=freq, image_size=imsize,
        )
```

```
    def render_egoposes_on_map(self, log_location: str, scene_tokens: List = None,
        self.explorer.render_egoposes_on_map(log_location, scene_tokens, out_path=c
```

In [30]:

```
class LyftDatasetExplorer:
    """Helper class to list and visualize Lyft Dataset data. These are meant to ser
    working with the data."""

    def __init__(self, lyftd: LyftDataset):
        self.lyftd = lyftd

    @staticmethod
    def get_color(category_name: str) -> Tuple[int, int, int]:
        """Provides the default colors based on the category names.
        This method works for the general Lyft Dataset categories, as well as the L
        Args:
            category_name:
        Returns:
        """
        if "bicycle" in category_name or "motorcycle" in category_name:
            return 255, 61, 99  # Red
        elif "vehicle" in category_name or category_name in ["bus", "car", "constru
            return 255, 158, 0  # Orange
        elif "pedestrian" in category_name:
            return 0, 0, 230  # Blue
        elif "cone" in category_name or "barrier" in category_name:
            return 0, 0, 0  # Black
        else:
            return 255, 0, 255  # Magenta

    def list_categories(self) -> None:
        """Print categories, counts and stats."""

        print("Category stats")

        # Add all annotations
        categories = dict()
        for record in self.lyftd.sample_annotation:
            if record["category_name"] not in categories:
                categories[record["category_name"]] = []
            categories[record["category_name"]].append(record["size"] + [record["si

        # Print stats
        for name, stats in sorted(categories.items()):
            stats = np.array(stats)
            print(
                "{:27} n={:5}, width={:5.2f}\u00B1{:.2f}, len={:5.2f}\u00B1{:.2f},
                "lw_aspect={:5.2f}\u00B1{:.2f}".format(
                    name[:27],
                    stats.shape[0],
                    np.mean(stats[:, 0]),
                    np.std(stats[:, 0]),
                    np.mean(stats[:, 1]),
                    np.std(stats[:, 1]),
                    np.mean(stats[:, 2]),
                    np.std(stats[:, 2]),
                    np.mean(stats[:, 3]),
                    np.std(stats[:, 3]),
                )
            )

    def list_attributes(self) -> None:
        """Prints attributes and counts."""
        attribute_counts = dict()
        for record in self.lyftd.sample_annotation:
```

```
            for attribute_token in record["attribute_tokens"]:
                att_name = self.lyftd.get("attribute", attribute_token)["name"]
                if att_name not in attribute_counts:
                    attribute_counts[att_name] = 0
                attribute_counts[att_name] += 1

        for name, count in sorted(attribute_counts.items()):
            print("{}: {}".format(name, count))

    def list_scenes(self) -> None:
        """ Lists all scenes with some meta data. """

        def ann_count(record):
            count = 0
            sample = self.lyftd.get("sample", record["first_sample_token"])
            while not sample["next"] == "":
                count += len(sample["anns"])
                sample = self.lyftd.get("sample", sample["next"])
            return count

        recs = [
            (self.lyftd.get("sample", record["first_sample_token"])["timestamp"], r
            for record in self.lyftd.scene
        ]

        for start_time, record in sorted(recs):
            start_time = self.lyftd.get("sample", record["first_sample_token"])["ti
            length_time = self.lyftd.get("sample", record["last_sample_token"])["ti
            location = self.lyftd.get("log", record["log_token"])["location"]
            desc = record["name"] + ", " + record["description"]
            if len(desc) > 55:
                desc = desc[:51] + "..."
            if len(location) > 18:
                location = location[:18]

            print(
                "{:16} [{}] {:4.0f}s, {}, #anns:{}".format(
                    desc,
                    datetime.utcfromtimestamp(start_time).strftime("%y-%m-%d %H:%M:
                    length_time,
                    location,
                    ann_count(record),
                )
            )

    def list_sample(self, sample_token: str) -> None:
        """Prints sample_data tokens and sample_annotation tokens related to the sa

        sample_record = self.lyftd.get("sample", sample_token)
        print("Sample: {}\n".format(sample_record["token"]))
        for sd_token in sample_record["data"].values():
            sd_record = self.lyftd.get("sample_data", sd_token)
            print(
                "sample_data_token: {}, mod: {}, channel: {}".format(
                    sd_token, sd_record["sensor_modality"], sd_record["channel"]
                )
            )
        print("")
        for ann_token in sample_record["anns"]:
            ann_record = self.lyftd.get("sample_annotation", ann_token)
            print("sample_annotation_token: {}, category: {}".format(ann_record["to

    def map_pointcloud_to_image(self, pointsensor_token: str, camera_token: str) ->
        """Given a point sensor (lidar/radar) token and camera sample_data token, l
```

```
        the image plane.
        Args:
            pointsensor_token: Lidar/radar sample_data token.
            camera_token: Camera sample_data token.
        Returns: (pointcloud <np.float: 2, n)>, coloring <np.float: n>, image <Imag
        """

        cam = self.lyftd.get("sample_data", camera_token)
        pointsensor = self.lyftd.get("sample_data", pointsensor_token)
        pcl_path = self.lyftd.data_path / ('train_' + pointsensor["filename"])
        if pointsensor["sensor_modality"] == "lidar":
            pc = LidarPointCloud.from_file(pcl_path)
        else:
            pc = RadarPointCloud.from_file(pcl_path)
        im = Image.open(str(self.lyftd.data_path / ('train_' + cam["filename"])))

        # Points live in the point sensor frame. So they need to be transformed via
        # First step: transform the point-cloud to the ego vehicle frame for the ti
        cs_record = self.lyftd.get("calibrated_sensor", pointsensor["calibrated_ser
        pc.rotate(Quaternion(cs_record["rotation"]).rotation_matrix)
        pc.translate(np.array(cs_record["translation"]))

        # Second step: transform to the global frame.
        poserecord = self.lyftd.get("ego_pose", pointsensor["ego_pose_token"])
        pc.rotate(Quaternion(poserecord["rotation"]).rotation_matrix)
        pc.translate(np.array(poserecord["translation"]))

        # Third step: transform into the ego vehicle frame for the timestamp of the
        poserecord = self.lyftd.get("ego_pose", cam["ego_pose_token"])
        pc.translate(-np.array(poserecord["translation"]))
        pc.rotate(Quaternion(poserecord["rotation"]).rotation_matrix.T)

        # Fourth step: transform into the camera.
        cs_record = self.lyftd.get("calibrated_sensor", cam["calibrated_sensor_toke
        pc.translate(-np.array(cs_record["translation"]))
        pc.rotate(Quaternion(cs_record["rotation"]).rotation_matrix.T)

        # Fifth step: actually take a "picture" of the point cloud.
        # Grab the depths (camera frame z axis points away from the camera).
        depths = pc.points[2, :]

        # Retrieve the color from the depth.
        coloring = depths

        # Take the actual picture (matrix multiplication with camera-matrix + renor
        points = view_points(pc.points[:3, :], np.array(cs_record["camera_intrinsic

        # Remove points that are either outside or behind the camera. Leave a margi
        mask = np.ones(depths.shape[0], dtype=bool)
        mask = np.logical_and(mask, depths > 0)
        mask = np.logical_and(mask, points[0, :] > 1)
        mask = np.logical_and(mask, points[0, :] < im.size[0] - 1)
        mask = np.logical_and(mask, points[1, :] > 1)
        mask = np.logical_and(mask, points[1, :] < im.size[1] - 1)
        points = points[:, mask]
        coloring = coloring[mask]

        return points, coloring, im

    def render_pointcloud_in_image(
        self,
        sample_token: str,
        dot_size: int = 2,
        pointsensor_channel: str = "LIDAR_TOP",
```

```
        camera_channel: str = "CAM_FRONT",
        out_path: str = None,
    ) -> None:
        """Scatter-plots a point-cloud on top of image.
        Args:
            sample_token: Sample token.
            dot_size: Scatter plot dot size.
            pointsensor_channel: RADAR or LIDAR channel name, e.g. 'LIDAR_TOP'.
            camera_channel: Camera channel name, e.g. 'CAM_FRONT'.
            out_path: Optional path to save the rendered figure to disk.
        Returns:
        """
        sample_record = self.lyftd.get("sample", sample_token)

        # Here we just grab the front camera and the point sensor.
        pointsensor_token = sample_record["data"][pointsensor_channel]
        camera_token = sample_record["data"][camera_channel]

        points, coloring, im = self.map_pointcloud_to_image(pointsensor_token, came
        plt.figure(figsize=(9, 16))
        plt.imshow(im)
        plt.scatter(points[0, :], points[1, :], c=coloring, s=dot_size)
        plt.axis("off")

        if out_path is not None:
            plt.savefig(out_path)

    def render_sample(
        self, token: str, box_vis_level: BoxVisibility = BoxVisibility.ANY, nsweeps
    ) -> None:
        """Render all LIDAR and camera sample_data in sample along with annotations
        Args:
            token: Sample token.
            box_vis_level: If sample_data is an image, this sets required visibilit
            nsweeps: Number of sweeps for lidar and radar.
            out_path: Optional path to save the rendered figure to disk.
        Returns:
        """
        record = self.lyftd.get("sample", token)

        # Separate RADAR from LIDAR and vision.
        radar_data = {}
        nonradar_data = {}
        for channel, token in record["data"].items():
            sd_record = self.lyftd.get("sample_data", token)
            sensor_modality = sd_record["sensor_modality"]
            if sensor_modality in ["lidar", "camera"]:
                nonradar_data[channel] = token
            else:
                radar_data[channel] = token

        num_radar_plots = 1 if len(radar_data) > 0 else 0

        # Create plots.
        n = num_radar_plots + len(nonradar_data)
        cols = 2
        fig, axes = plt.subplots(int(np.ceil(n / cols)), cols, figsize=(16, 24))

        if len(radar_data) > 0:
            # Plot radar into a single subplot.
            ax = axes[0, 0]
            for i, (_, sd_token) in enumerate(radar_data.items()):
                self.render_sample_data(
                    sd_token, with_anns=i == 0, box_vis_level=box_vis_level, ax=ax,
```

```
                )
            ax.set_title("Fused RADARs")

        # Plot camera and lidar in separate subplots.
        for (_, sd_token), ax in zip(nonradar_data.items(), axes.flatten()[num_rada
            self.render_sample_data(sd_token, box_vis_level=box_vis_level, ax=ax, r

        axes.flatten()[-1].axis("off")
        plt.tight_layout()
        fig.subplots_adjust(wspace=0, hspace=0)

        if out_path is not None:
            plt.savefig(out_path)

    def render_ego_centric_map(self, sample_data_token: str, axes_limit: float = 40
        """Render map centered around the associated ego pose.
        Args:
            sample_data_token: Sample_data token.
            axes_limit: Axes limit measured in meters.
            ax: Axes onto which to render.
        """

        def crop_image(image: np.array, x_px: int, y_px: int, axes_limit_px: int) -
            x_min = int(x_px - axes_limit_px)
            x_max = int(x_px + axes_limit_px)
            y_min = int(y_px - axes_limit_px)
            y_max = int(y_px + axes_limit_px)

            cropped_image = image[y_min:y_max, x_min:x_max]

            return cropped_image

        sd_record = self.lyftd.get("sample_data", sample_data_token)

        # Init axes.
        if ax is None:
            _, ax = plt.subplots(1, 1, figsize=(9, 9))

        sample = self.lyftd.get("sample", sd_record["sample_token"])
        scene = self.lyftd.get("scene", sample["scene_token"])
        log = self.lyftd.get("log", scene["log_token"])
        map = self.lyftd.get("map", log["map_token"])
        map_mask = map["mask"]

        pose = self.lyftd.get("ego_pose", sd_record["ego_pose_token"])
        pixel_coords = map_mask.to_pixel_coords(pose["translation"][0], pose["trans

        scaled_limit_px = int(axes_limit * (1.0 / map_mask.resolution))
        mask_raster = map_mask.mask()

        cropped = crop_image(mask_raster, pixel_coords[0], pixel_coords[1], int(sca

        ypr_rad = Quaternion(pose["rotation"]).yaw_pitch_roll
        yaw_deg = -math.degrees(ypr_rad[0])

        rotated_cropped = np.array(Image.fromarray(cropped).rotate(yaw_deg))
        ego_centric_map = crop_image(
            rotated_cropped, rotated_cropped.shape[1] / 2, rotated_cropped.shape[0]
        )
        ax.imshow(
            ego_centric_map, extent=[-axes_limit, axes_limit, -axes_limit, axes_lim
        )

    def render_sample_data(
```

```
        self,
        sample_data_token: str,
        with_anns: bool = True,
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        axes_limit: float = 40,
        ax: Axes = None,
        num_sweeps: int = 1,
        out_path: str = None,
        underlay_map: bool = False,
    ):
        """Render sample data onto axis.
        Args:
            sample_data_token: Sample_data token.
            with_anns: Whether to draw annotations.
            box_vis_level: If sample_data is an image, this sets required visibilit
            axes_limit: Axes limit for lidar and radar (measured in meters).
            ax: Axes onto which to render.
            num_sweeps: Number of sweeps for lidar and radar.
            out_path: Optional path to save the rendered figure to disk.
            underlay_map: When set to true, LIDAR data is plotted onto the map. Thi
        """

        # Get sensor modality.
        sd_record = self.lyftd.get("sample_data", sample_data_token)
        sensor_modality = sd_record["sensor_modality"]

        if sensor_modality == "lidar":
            # Get boxes in lidar frame.
            _, boxes, _ = self.lyftd.get_sample_data(
                sample_data_token, box_vis_level=box_vis_level, flat_vehicle_coordi
            )

            # Get aggregated point cloud in lidar frame.
            sample_rec = self.lyftd.get("sample", sd_record["sample_token"])
            chan = sd_record["channel"]
            ref_chan = "LIDAR_TOP"
            pc, times = LidarPointCloud.from_file_multisweep(
                self.lyftd, sample_rec, chan, ref_chan, num_sweeps=num_sweeps
            )

            # Compute transformation matrices for lidar point cloud
            cs_record = self.lyftd.get("calibrated_sensor", sd_record["calibrated_s
            pose_record = self.lyftd.get("ego_pose", sd_record["ego_pose_token"])
            vehicle_from_sensor = np.eye(4)
            vehicle_from_sensor[:3, :3] = Quaternion(cs_record["rotation"]).rotatic
            vehicle_from_sensor[:3, 3] = cs_record["translation"]

            ego_yaw = Quaternion(pose_record["rotation"]).yaw_pitch_roll[0]
            rot_vehicle_flat_from_vehicle = np.dot(
                Quaternion(scalar=np.cos(ego_yaw / 2), vector=[0, 0, np.sin(ego_yaw
                Quaternion(pose_record["rotation"]).inverse.rotation_matrix,
            )

            vehicle_flat_from_vehicle = np.eye(4)
            vehicle_flat_from_vehicle[:3, :3] = rot_vehicle_flat_from_vehicle

            # Init axes.
            if ax is None:
                _, ax = plt.subplots(1, 1, figsize=(9, 9))

            if underlay_map:
                self.render_ego_centric_map(sample_data_token=sample_data_token, ax

            # Show point cloud.
```

```
            points = view_points(
                pc.points[:3, :], np.dot(vehicle_flat_from_vehicle, vehicle_from_se
            )
            dists = np.sqrt(np.sum(pc.points[:2, :] ** 2, axis=0))
            colors = np.minimum(1, dists / axes_limit / np.sqrt(2))
            ax.scatter(points[0, :], points[1, :], c=colors, s=0.2)

            # Show ego vehicle.
            ax.plot(0, 0, "x", color="red")

            # Show boxes.
            if with_anns:
                for box in boxes:
                    c = np.array(self.get_color(box.name)) / 255.0
                    box.render(ax, view=np.eye(4), colors=(c, c, c))

            # Limit visible range.
            ax.set_xlim(-axes_limit, axes_limit)
            ax.set_ylim(-axes_limit, axes_limit)

        elif sensor_modality == "radar":
            # Get boxes in lidar frame.
            sample_rec = self.lyftd.get("sample", sd_record["sample_token"])
            lidar_token = sample_rec["data"]["LIDAR_TOP"]
            _, boxes, _ = self.lyftd.get_sample_data(lidar_token, box_vis_level=box

            # Get aggregated point cloud in lidar frame.
            # The point cloud is transformed to the lidar frame for visualization p
            chan = sd_record["channel"]
            ref_chan = "LIDAR_TOP"
            pc, times = RadarPointCloud.from_file_multisweep(
                self.lyftd, sample_rec, chan, ref_chan, num_sweeps=num_sweeps
            )

            # Transform radar velocities (x is front, y is left), as these are not
            # cloud.
            radar_cs_record = self.lyftd.get("calibrated_sensor", sd_record["calibr
            lidar_sd_record = self.lyftd.get("sample_data", lidar_token)
            lidar_cs_record = self.lyftd.get("calibrated_sensor", lidar_sd_record['
            velocities = pc.points[8:10, :]  # Compensated velocity
            velocities = np.vstack((velocities, np.zeros(pc.points.shape[1])))
            velocities = np.dot(Quaternion(radar_cs_record["rotation"]).rotation_ma
            velocities = np.dot(Quaternion(lidar_cs_record["rotation"]).rotation_ma
            velocities[2, :] = np.zeros(pc.points.shape[1])

            # Init axes.
            if ax is None:
                _, ax = plt.subplots(1, 1, figsize=(9, 9))

            # Show point cloud.
            points = view_points(pc.points[:3, :], np.eye(4), normalize=False)
            dists = np.sqrt(np.sum(pc.points[:2, :] ** 2, axis=0))
            colors = np.minimum(1, dists / axes_limit / np.sqrt(2))
            sc = ax.scatter(points[0, :], points[1, :], c=colors, s=3)

            # Show velocities.
            points_vel = view_points(pc.points[:3, :] + velocities, np.eye(4), norm
            max_delta = 10
            deltas_vel = points_vel - points
            deltas_vel = 3 * deltas_vel  # Arbitrary scaling
            deltas_vel = np.clip(deltas_vel, -max_delta, max_delta)  # Arbitrary cl
            colors_rgba = sc.to_rgba(colors)
            for i in range(points.shape[1]):
                ax.arrow(points[0, i], points[1, i], deltas_vel[0, i], deltas_vel[1
```

```
            # Show ego vehicle.
            ax.plot(0, 0, "x", color="black")

            # Show boxes.
            if with_anns:
                for box in boxes:
                    c = np.array(self.get_color(box.name)) / 255.0
                    box.render(ax, view=np.eye(4), colors=(c, c, c))

            # Limit visible range.
            ax.set_xlim(-axes_limit, axes_limit)
            ax.set_ylim(-axes_limit, axes_limit)

        elif sensor_modality == "camera":
            # Load boxes and image.
            data_path, boxes, camera_intrinsic = self.lyftd.get_sample_data(
                sample_data_token, box_vis_level=box_vis_level
            )

            data = Image.open(str(data_path)[:len(str(data_path)) - 46] + 'train_im
                              str(data_path)[len(str(data_path)) - 39 : len(str(dat

            # Init axes.
            if ax is None:
                _, ax = plt.subplots(1, 1, figsize=(9, 16))

            # Show image.
            ax.imshow(data)

            # Show boxes.
            if with_anns:
                for box in boxes:
                    c = np.array(self.get_color(box.name)) / 255.0
                    box.render(ax, view=camera_intrinsic, normalize=True, colors=(c

            # Limit visible range.
            ax.set_xlim(0, data.size[0])
            ax.set_ylim(data.size[1], 0)

        else:
            raise ValueError("Error: Unknown sensor modality!")

        ax.axis("off")
        ax.set_title(sd_record["channel"])
        ax.set_aspect("equal")

        if out_path is not None:
            num = len([name for name in os.listdir(out_path)])
            out_path = out_path + str(num).zfill(5) + "_" + sample_data_token + ".p
            plt.savefig(out_path)
            plt.close("all")
            return out_path

    def render_annotation(
        self,
        ann_token: str,
        margin: float = 10,
        view: np.ndarray = np.eye(4),
        box_vis_level: BoxVisibility = BoxVisibility.ANY,
        out_path: str = None,
    ) -> None:
        """Render selected annotation.
        Args:
```

```
            ann_token: Sample_annotation token.
            margin: How many meters in each direction to include in LIDAR view.
            view: LIDAR view point.
            box_vis_level: If sample_data is an image, this sets required visibilit
            out_path: Optional path to save the rendered figure to disk.
        """

        ann_record = self.lyftd.get("sample_annotation", ann_token)
        sample_record = self.lyftd.get("sample", ann_record["sample_token"])
        assert "LIDAR_TOP" in sample_record["data"].keys(), "No LIDAR_TOP in data,

        fig, axes = plt.subplots(1, 2, figsize=(18, 9))

        # Figure out which camera the object is fully visible in (this may return n
        boxes, cam = [], []
        cams = [key for key in sample_record["data"].keys() if "CAM" in key]
        for cam in cams:
            _, boxes, _ = self.lyftd.get_sample_data(
                sample_record["data"][cam], box_vis_level=box_vis_level, selected_a
            )
            if len(boxes) > 0:
                break  # We found an image that matches. Let's abort.
        assert len(boxes) > 0, "Could not find image where annotation is visible. T
        assert len(boxes) < 2, "Found multiple annotations. Something is wrong!"

        cam = sample_record["data"][cam]

        # Plot LIDAR view
        lidar = sample_record["data"]["LIDAR_TOP"]
        data_path, boxes, camera_intrinsic = self.lyftd.get_sample_data(lidar, sele
        LidarPointCloud.from_file(Path(str(data_path)[:len(str(data_path)) - 46] +
                                       str(data_path)[len(str(data_path)) - 40 : le
        for box in boxes:
            c = np.array(self.get_color(box.name)) / 255.0
            box.render(axes[0], view=view, colors=(c, c, c))
            corners = view_points(boxes[0].corners(), view, False)[:2, :]
            axes[0].set_xlim([np.min(corners[0, :]) - margin, np.max(corners[0, :])
            axes[0].set_ylim([np.min(corners[1, :]) - margin, np.max(corners[1, :])
            axes[0].axis("off")
            axes[0].set_aspect("equal")

        # Plot CAMERA view
        data_path, boxes, camera_intrinsic = self.lyftd.get_sample_data(cam, select
        im = Image.open(Path(str(data_path)[:len(str(data_path)) - 46] + 'train_ima
                             str(data_path)[len(str(data_path)) - 39 : len(str(data
        axes[1].imshow(im)
        axes[1].set_title(self.lyftd.get("sample_data", cam)["channel"])
        axes[1].axis("off")
        axes[1].set_aspect("equal")
        for box in boxes:
            c = np.array(self.get_color(box.name)) / 255.0
            box.render(axes[1], view=camera_intrinsic, normalize=True, colors=(c, c

        if out_path is not None:
            plt.savefig(out_path)

    def render_instance(self, instance_token: str, out_path: str = None) -> None:
        """Finds the annotation of the given instance that is closest to the vehicl
        Args:
            instance_token: The instance token.
            out_path: Optional path to save the rendered figure to disk.
        Returns:
        """
```

```
        ann_tokens = self.lyftd.field2token("sample_annotation", "instance_token",
        closest = [np.inf, None]
        for ann_token in ann_tokens:
            ann_record = self.lyftd.get("sample_annotation", ann_token)
            sample_record = self.lyftd.get("sample", ann_record["sample_token"])
            sample_data_record = self.lyftd.get("sample_data", sample_record["data'
            pose_record = self.lyftd.get("ego_pose", sample_data_record["ego_pose_t
            dist = np.linalg.norm(np.array(pose_record["translation"]) - np.array(a
            if dist < closest[0]:
                closest[0] = dist
                closest[1] = ann_token
        self.render_annotation(closest[1], out_path=out_path)

    def render_scene(self, scene_token: str, freq: float = 10, image_width: int = 6
        """Renders a full scene with all surround view camera channels.
        Args:
            scene_token: Unique identifier of scene to render.
            freq: Display frequency (Hz).
            image_width: Width of image to render. Height is determined automatical
            out_path: Optional path to write a video file of the rendered frames.
        """

        if out_path is not None:
            assert out_path.suffix == ".avi"

        # Get records from DB.
        scene_rec = self.lyftd.get("scene", scene_token)
        first_sample_rec = self.lyftd.get("sample", scene_rec["first_sample_token"]
        last_sample_rec = self.lyftd.get("sample", scene_rec["last_sample_token"])

        channels = ["CAM_FRONT_LEFT", "CAM_FRONT", "CAM_FRONT_RIGHT", "CAM_BACK_LEF

        horizontal_flip = ["CAM_BACK_LEFT", "CAM_BACK", "CAM_BACK_RIGHT"]  # Flip t

        time_step = 1 / freq * 1e6  # Time-stamps are measured in micro-seconds.

        window_name = "{}".format(scene_rec["name"])
        cv2.namedWindow(window_name)
        cv2.moveWindow(window_name, 0, 0)

        # Load first sample_data record for each channel
        current_recs = {}  # Holds the current record to be displayed by channel.
        prev_recs = {}  # Hold the previous displayed record by channel.
        for channel in channels:
            current_recs[channel] = self.lyftd.get("sample_data", first_sample_rec[
            prev_recs[channel] = None

        # We assume that the resolution is the same for all surround view cameras.
        image_height = int(image_width * current_recs[channels[0]]["height"] / curr
        image_size = (image_width, image_height)

        # Set some display parameters
        layout = {
            "CAM_FRONT_LEFT": (0, 0),
            "CAM_FRONT": (image_size[0], 0),
            "CAM_FRONT_RIGHT": (2 * image_size[0], 0),
            "CAM_BACK_LEFT": (0, image_size[1]),
            "CAM_BACK": (image_size[0], image_size[1]),
            "CAM_BACK_RIGHT": (2 * image_size[0], image_size[1]),
        }

        canvas = np.ones((2 * image_size[1], 3 * image_size[0], 3), np.uint8)
        if out_path is not None:
            fourcc = cv2.VideoWriter_fourcc(*"MJPG")
```

```
            out = cv2.VideoWriter(out_path, fourcc, freq, canvas.shape[1::-1])
        else:
            out = None

        current_time = first_sample_rec["timestamp"]

        while current_time < last_sample_rec["timestamp"]:

            current_time += time_step

            # For each channel, find first sample that has time > current_time.
            for channel, sd_rec in current_recs.items():
                while sd_rec["timestamp"] < current_time and sd_rec["next"] != "":
                    sd_rec = self.lyftd.get("sample_data", sd_rec["next"])
                    current_recs[channel] = sd_rec

            # Now add to canvas
            for channel, sd_rec in current_recs.items():

                # Only update canvas if we have not already rendered this one.
                if not sd_rec == prev_recs[channel]:

                    # Get annotations and params from DB.
                    image_path, boxes, camera_intrinsic = self.lyftd.get_sample_dat
                        sd_rec["token"], box_vis_level=BoxVisibility.ANY
                    )

                    # Load and render
                    if not image_path.exists():
                        raise Exception("Error: Missing image %s" % image_path)
                    im = cv2.imread(str(image_path))
                    for box in boxes:
                        c = self.get_color(box.name)
                        box.render_cv2(im, view=camera_intrinsic, normalize=True, c

                    im = cv2.resize(im, image_size)
                    if channel in horizontal_flip:
                        im = im[:, ::-1, :]

                    canvas[
                        layout[channel][1] : layout[channel][1] + image_size[1],
                        layout[channel][0] : layout[channel][0] + image_size[0],
                        :,
                    ] = im

                    prev_recs[channel] = sd_rec  # Store here so we don't render th

            # Show updated canvas.
            cv2.imshow(window_name, canvas)
            if out_path is not None:
                out.write(canvas)

            key = cv2.waitKey(1)  # Wait a very short time (1 ms).

            if key == 32:  # if space is pressed, pause.
                key = cv2.waitKey()

            if key == 27:  # if ESC is pressed, exit.
                cv2.destroyAllWindows()
                break

        cv2.destroyAllWindows()
        if out_path is not None:
            out.release()
```

```python
    def render_scene_channel(
        self,
        scene_token: str,
        channel: str = "CAM_FRONT",
        freq: float = 10,
        image_size: Tuple[float, float] = (640, 360),
        out_path: Path = None,
    ) -> None:
        """Renders a full scene for a particular camera channel.
        Args:
            scene_token: Unique identifier of scene to render.
            channel: Channel to render.
            freq: Display frequency (Hz).
            image_size: Size of image to render. The larger the slower this will ru
            out_path: Optional path to write a video file of the rendered frames.
        """

        valid_channels = [
            "CAM_FRONT_LEFT",
            "CAM_FRONT",
            "CAM_FRONT_RIGHT",
            "CAM_BACK_LEFT",
            "CAM_BACK",
            "CAM_BACK_RIGHT",
        ]

        assert image_size[0] / image_size[1] == 16 / 9, "Aspect ratio should be 16/
        assert channel in valid_channels, "Input channel {} not valid.".format(char

        if out_path is not None:
            assert out_path.suffix == ".avi"

        # Get records from DB
        scene_rec = self.lyftd.get("scene", scene_token)
        sample_rec = self.lyftd.get("sample", scene_rec["first_sample_token"])
        sd_rec = self.lyftd.get("sample_data", sample_rec["data"][channel])

        # Open CV init
        name = "{}: {} (Space to pause, ESC to exit)".format(scene_rec["name"], cha
        cv2.namedWindow(name)
        cv2.moveWindow(name, 0, 0)

        if out_path is not None:
            fourcc = cv2.VideoWriter_fourcc(*"MJPG")
            out = cv2.VideoWriter(out_path, fourcc, freq, image_size)
        else:
            out = None

        has_more_frames = True
        while has_more_frames:

            # Get data from DB
            image_path, boxes, camera_intrinsic = self.lyftd.get_sample_data(
                sd_rec["token"], box_vis_level=BoxVisibility.ANY
            )

            # Load and render
            if not image_path.exists():
                raise Exception("Error: Missing image %s" % image_path)
            image = cv2.imread(str(image_path))
            for box in boxes:
                c = self.get_color(box.name)
                box.render_cv2(image, view=camera_intrinsic, normalize=True, colors
```

```
            # Render
            image = cv2.resize(image, image_size)
            cv2.imshow(name, image)
            if out_path is not None:
                out.write(image)

            key = cv2.waitKey(10)  # Images stored at approx 10 Hz, so wait 10 ms.
            if key == 32:  # If space is pressed, pause.
                key = cv2.waitKey()

            if key == 27:  # if ESC is pressed, exit
                cv2.destroyAllWindows()
                break

            if not sd_rec["next"] == "":
                sd_rec = self.lyftd.get("sample_data", sd_rec["next"])
            else:
                has_more_frames = False

        cv2.destroyAllWindows()
        if out_path is not None:
            out.release()

    def render_egoposes_on_map(
        self,
        log_location: str,
        scene_tokens: List = None,
        close_dist: float = 100,
        color_fg: Tuple[int, int, int] = (167, 174, 186),
        color_bg: Tuple[int, int, int] = (255, 255, 255),
        out_path: Path = None,
    ) -> None:
        """Renders ego poses a the map. These can be filtered by location or scene.
        Args:
            log_location: Name of the location, e.g. "singapore-onenorth", "singapo
                          "singapore-queenstown' and "boston-seaport".
            scene_tokens: Optional list of scene tokens.
            close_dist: Distance in meters for an ego pose to be considered within
            color_fg: Color of the semantic prior in RGB format (ignored if map is
            color_bg: Color of the non-semantic prior in RGB format (ignored if map
            out_path: Optional path to save the rendered figure to disk.
        Returns:
        """

        # Get logs by location
        log_tokens = [l["token"] for l in self.lyftd.log if l["location"] == log_lo
        assert len(log_tokens) > 0, "Error: This split has 0 scenes for location %s

        # Filter scenes
        scene_tokens_location = [e["token"] for e in self.lyftd.scene if e["log_tok
        if scene_tokens is not None:
            scene_tokens_location = [t for t in scene_tokens_location if t in scene
        if len(scene_tokens_location) == 0:
            print("Warning: Found 0 valid scenes for location %s!" % log_location)

        map_poses = []
        map_mask = None

        print("Adding ego poses to map...")
        for scene_token in tqdm(scene_tokens_location):

            # Get records from the database.
            scene_record = self.lyftd.get("scene", scene_token)
```

```
            log_record = self.lyftd.get("log", scene_record["log_token"])
            map_record = self.lyftd.get("map", log_record["map_token"])
            map_mask = map_record["mask"]

            # For each sample in the scene, store the ego pose.
            sample_tokens = self.lyftd.field2token("sample", "scene_token", scene_t
            for sample_token in sample_tokens:
                sample_record = self.lyftd.get("sample", sample_token)

                # Poses are associated with the sample_data. Here we use the lidar
                sample_data_record = self.lyftd.get("sample_data", sample_record["d
                pose_record = self.lyftd.get("ego_pose", sample_data_record["ego_po

                # Calculate the pose on the map and append
                map_poses.append(
                    np.concatenate(
                        map_mask.to_pixel_coords(pose_record["translation"][0], pos
                    )
                )

        # Compute number of close ego poses.
        print("Creating plot...")
        map_poses = np.vstack(map_poses)
        dists = sklearn.metrics.pairwise.euclidean_distances(map_poses * map_mask.r
        close_poses = np.sum(dists < close_dist, axis=0)

        if len(np.array(map_mask.mask()).shape) == 3 and np.array(map_mask.mask()).
            # RGB Colour maps
            mask = map_mask.mask()
        else:
            # Monochrome maps
            # Set the colors for the mask.
            mask = Image.fromarray(map_mask.mask())
            mask = np.array(mask)

            maskr = color_fg[0] * np.ones(np.shape(mask), dtype=np.uint8)
            maskr[mask == 0] = color_bg[0]
            maskg = color_fg[1] * np.ones(np.shape(mask), dtype=np.uint8)
            maskg[mask == 0] = color_bg[1]
            maskb = color_fg[2] * np.ones(np.shape(mask), dtype=np.uint8)
            maskb[mask == 0] = color_bg[2]
            mask = np.concatenate(
                (np.expand_dims(maskr, axis=2), np.expand_dims(maskg, axis=2), np.e
            )

        # Plot.
        _, ax = plt.subplots(1, 1, figsize=(10, 10))
        ax.imshow(mask)
        title = "Number of ego poses within {}m in {}".format(close_dist, log_locat
        ax.set_title(title, color="k")
        sc = ax.scatter(map_poses[:, 0], map_poses[:, 1], s=10, c=close_poses)
        color_bar = plt.colorbar(sc, fraction=0.025, pad=0.04)
        plt.rcParams["figure.facecolor"] = "black"
        color_bar_ticklabels = plt.getp(color_bar.ax.axes, "yticklabels")
        plt.setp(color_bar_ticklabels, color="k")
        plt.rcParams["figure.facecolor"] = "white"  # Reset for future plots

        if out_path is not None:
            plt.savefig(out_path)
            plt.close("all")
```

In [31]:
```
lyft_dataset = LyftDataset(data_path=DATA_PATH, json_path=DATA_PATH+'train_data')
```

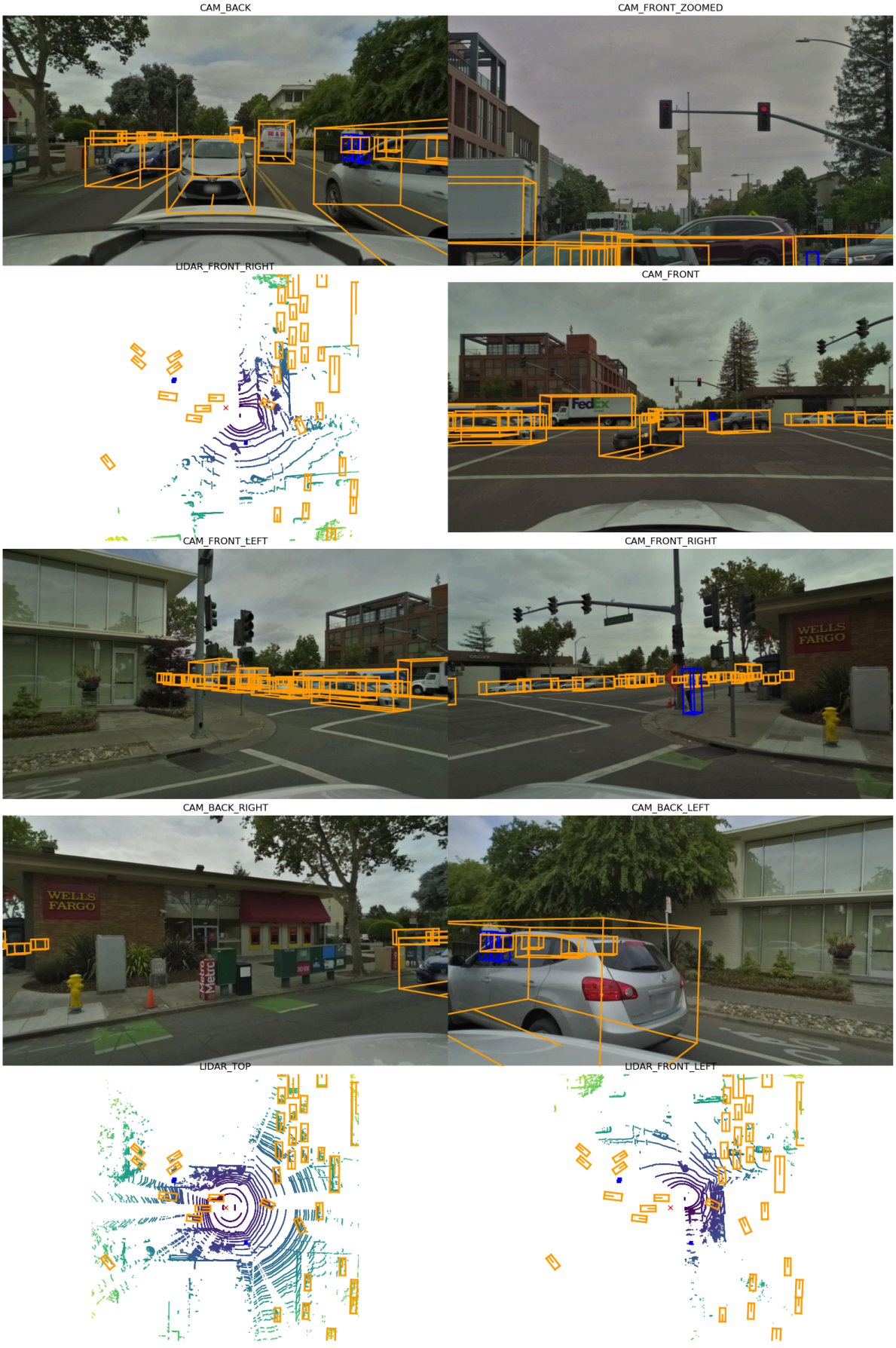
```
9 category,
18 attribute,
4 visibility,
18421 instance,
10 sensor,
148 calibrated_sensor,
177789 ego_pose,
180 log,
180 scene,
22680 sample,
189504 sample_data,
638179 sample_annotation,
1 map,
Done loading in 21.9 seconds.
======
Reverse indexing ...
Done reverse indexing in 6.0 seconds.
======
```

In [32]:
```
my_scene = lyft_dataset.scene[0]
my_scene
```

Out[32]:
```
{'log_token': 'da4ed9e02f64c544f4f1f10c6738216dcb0e6b0d50952e158e5589854af9f100',
 'first_sample_token': '24b0962e44420e6322de3f25d9e4e5cc3c7a348ec00bfa69db21517e4ca92cc8',
 'name': 'host-a101-lidar0-1241893239199111666-1241893264098084346',
 'description': '',
 'last_sample_token': '2346756c83f6ae8c4d1adec62b4d0d31b62116d2e1819e96e9512667d15e7cec',
 'nbr_samples': 126,
 'token': 'da4ed9e02f64c544f4f1f10c6738216dcb0e6b0d50952e158e5589854af9f100'}
```

In [33]:
```
def render_scene(index):
    my_scene = lyft_dataset.scene[index]
    my_sample_token = my_scene["first_sample_token"]
    lyft_dataset.render_sample(my_sample_token)
```

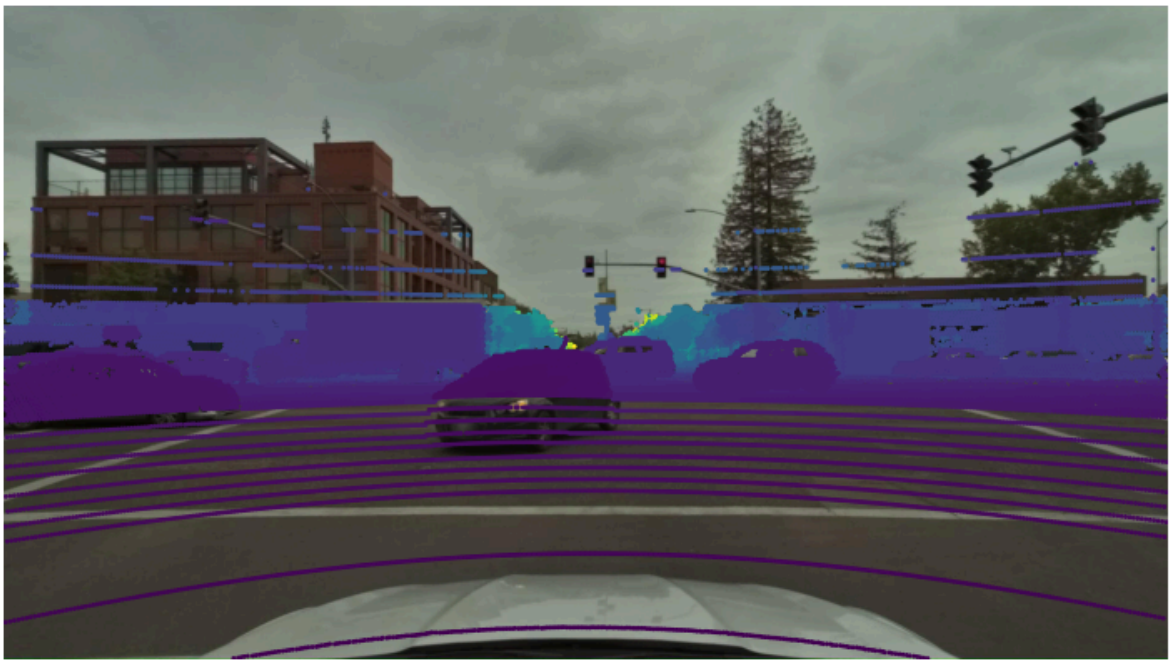
In [34]:
```
render_scene(0)
```



In [35]:
```
render_scene(1)
```

In [36]:
```
my_sample_token = my_scene["first_sample_token"]
my_sample = lyft_dataset.get('sample', my_sample_token)
```

In [37]:
```
lyft_dataset.render_pointcloud_in_image(sample_token = my_sample["token"],
                                        dot_size = 1,
                                        camera_channel = 'CAM_FRONT')
```



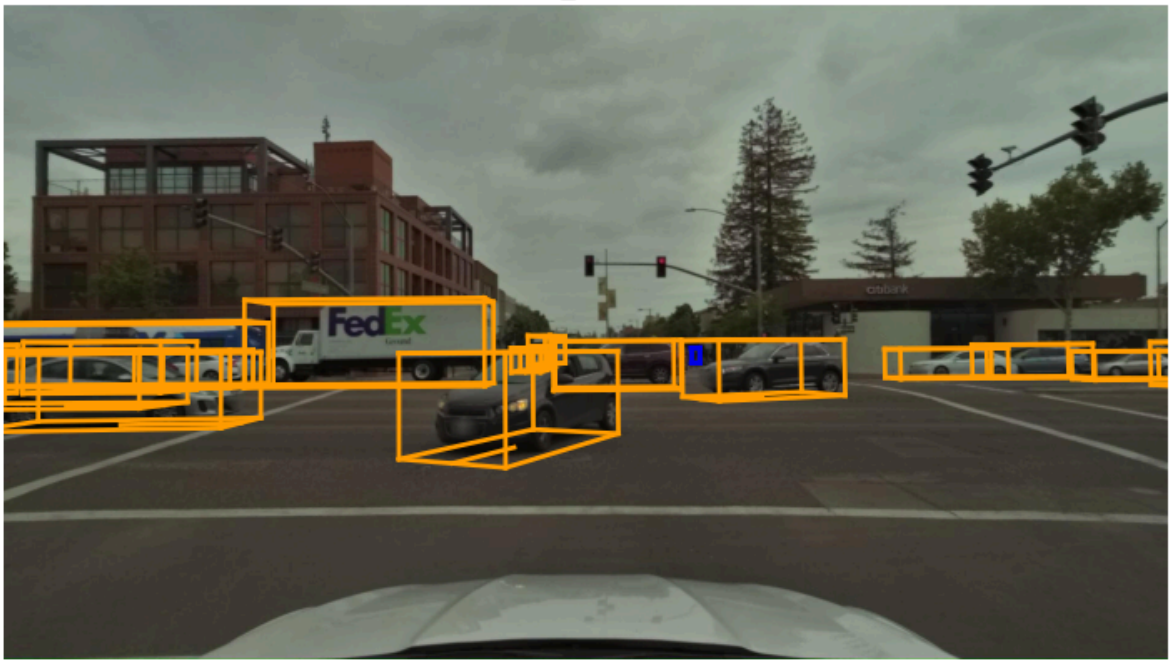
In [38]:
```
my_sample['data']
```

Out[38]:
```
{'CAM_BACK': '542a9e44f2e26221a6aa767c2a9b90a9f692c3aee2edb7145256b61e666633a4',
 'CAM_FRONT_ZOOMED': '9c9bc711d93d728666f5d7499703624249919dd1b290a477fcfa39f41b26
259e',
 'LIDAR_FRONT_RIGHT': '8cfae06bc3d5d7f9be081f66157909ff18c9f332cc173d962460239990c
7a4ff',
 'CAM_FRONT': 'fb40b3b5b9d289cd0e763bec34e327d3317a7b416f787feac0d387363b4d00f0',
 'CAM_FRONT_LEFT': 'f47a5d143bcebb24efc269b1a40ecb09440003df2c381a69e67cd2a726b27a
0c',
 'CAM_FRONT_RIGHT': '5dc54375a9e14e8398a538ff97fbbee7543b6f5df082c60fc4477c919ba83
a40',
 'CAM_BACK_RIGHT': 'ae8754c733560aa2506166cfaf559aeba670407631badadb065a9ffe7c337a
7d',
 'CAM_BACK_LEFT': '01c0eecd4b56668e949143e02a117b5683025766d186920099d1e918c23c8b4
b',
 'LIDAR_TOP': 'ec9950f7b5d4ae85ae48d07786e09cebbf4ee771d054353f1e24a95700b4c4af',
 'LIDAR_FRONT_LEFT': '5c3d79e1cf8c8182b2ceefa33af96cbebfc71f92e18bf64eb8d4e0bf162e
01d4'}
```

In [39]:
```
sensor_channel = 'CAM_FRONT'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
```

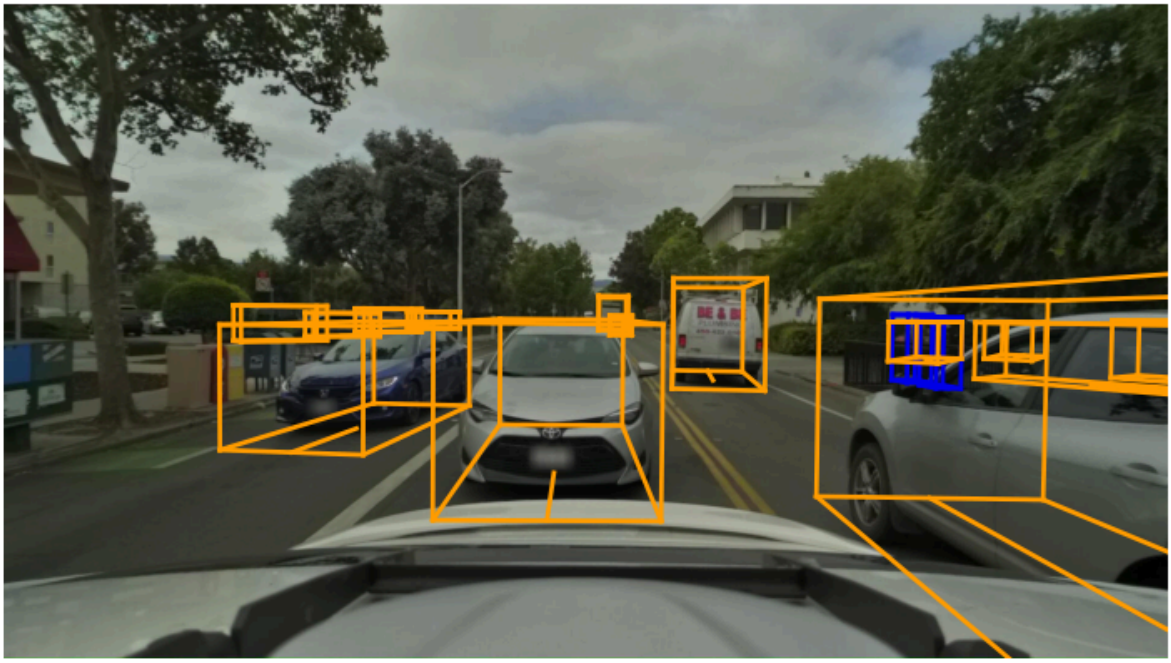
In [40]:
```
lyft_dataset.render_sample_data(my_sample_data['token'])
```

CAM_FRONT



In [41]:
```
sensor_channel = 'CAM_BACK'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
lyft_dataset.render_sample_data(my_sample_data['token'])
```

CAM_BACK

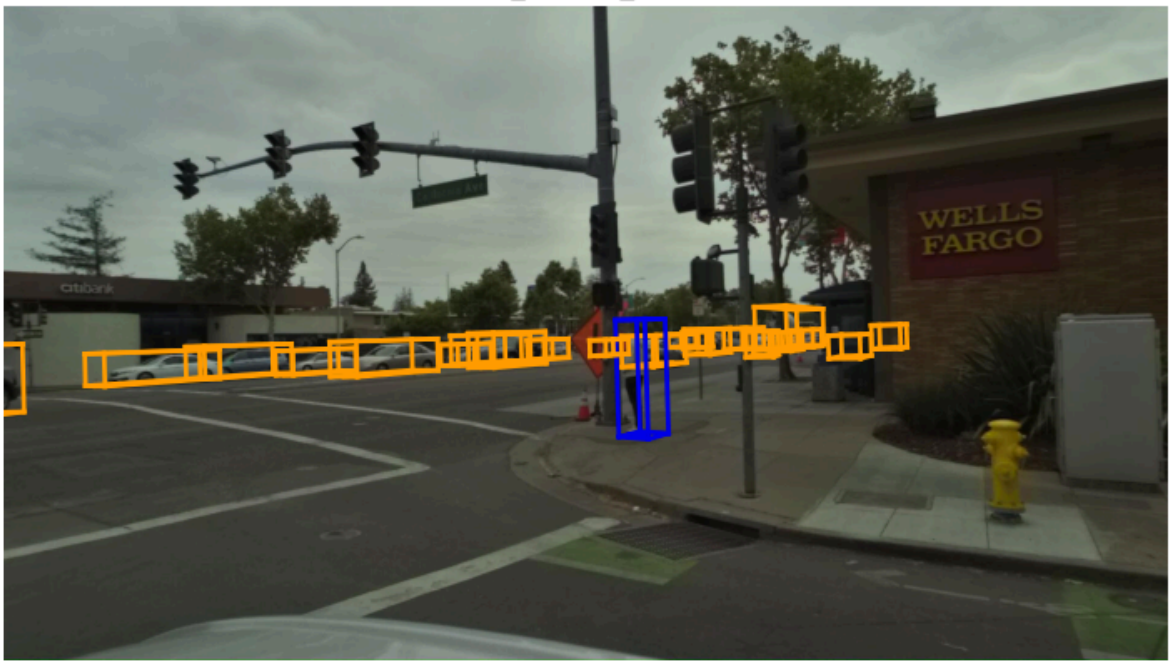


In [42]:
```
sensor_channel = 'CAM_FRONT_LEFT'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
lyft_dataset.render_sample_data(my_sample_data['token'])
```

CAM_FRONT_LEFT

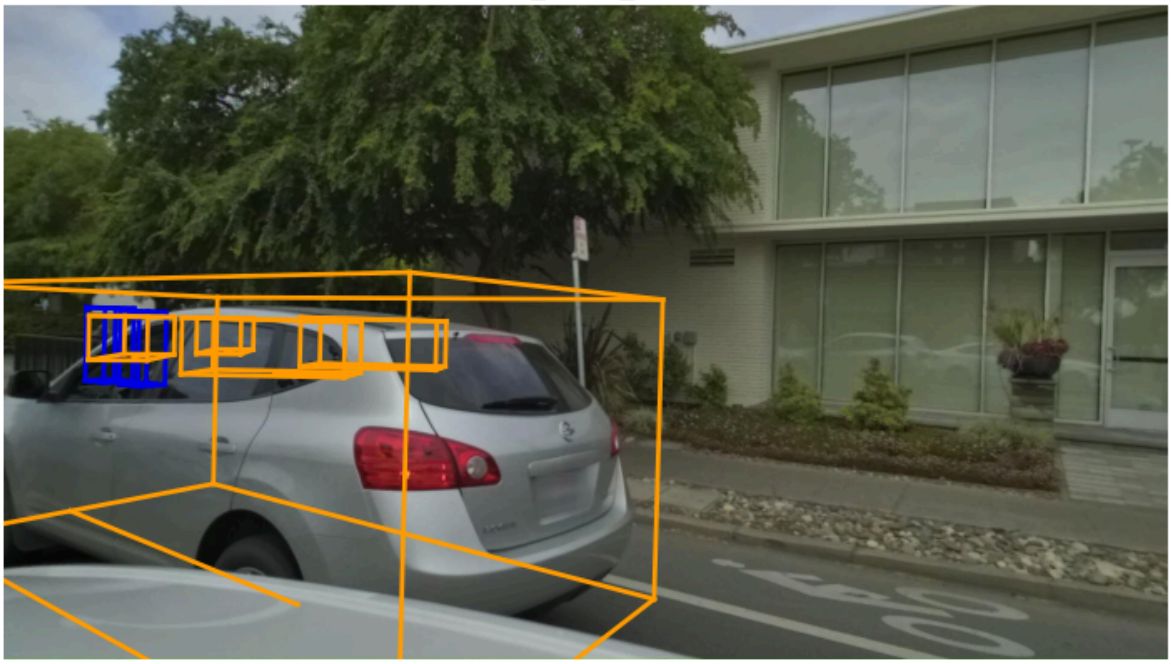
In [43]:
```
sensor_channel = 'CAM_FRONT_RIGHT'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
lyft_dataset.render_sample_data(my_sample_data['token'])
```

CAM_FRONT_RIGHT



In [44]:
```
sensor_channel = 'CAM_BACK_LEFT'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
lyft_dataset.render_sample_data(my_sample_data['token'])
```

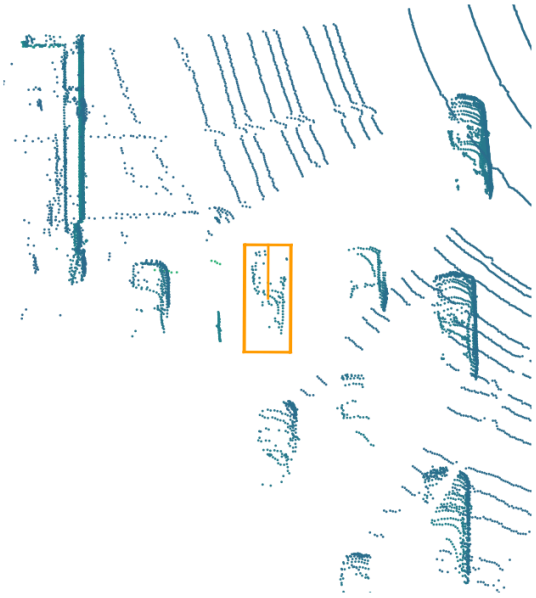
CAM_BACK_LEFT



In [45]:
```
sensor_channel = 'CAM_BACK_RIGHT'
my_sample_data = lyft_dataset.get('sample_data', my_sample['data'][sensor_channel])
lyft_dataset.render_sample_data(my_sample_data['token'])
```

CAM_BACK_RIGHT

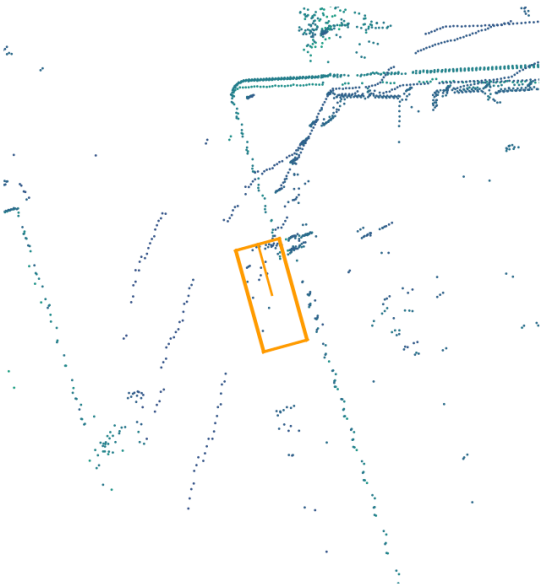
In [46]:
```
my_annotation_token = my_sample['anns'][10]
my_annotation =  my_sample_data.get('sample_annotation', my_annotation_token)
lyft_dataset.render_annotation(my_annotation_token)
```



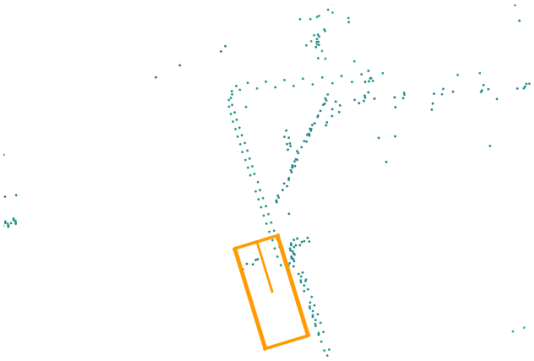
In [47]:
```
my_instance = lyft_dataset.instance[100]
my_instance
```

Out[47]:
```
{'last_annotation_token': '17be5c0175d3f4e36b9829d4789a6174d531637587107a0bfa1f3668d1ed9a57',
 'category_token': '8eccddb83fa7f8f992b2500f2ad658f65c9095588f3bc0ae338d97aff2dbcb9c',
 'token': '0984d7f7d64a8a03523e43ea01c7292488bd73ae745514077b28a9dc240b9f95',
 'first_annotation_token': 'b1f924cb04786ef4d46810ef8c3e7fe17597d4464507267dc56dcfbe1faa6581',
 'nbr_annotations': 56}
```

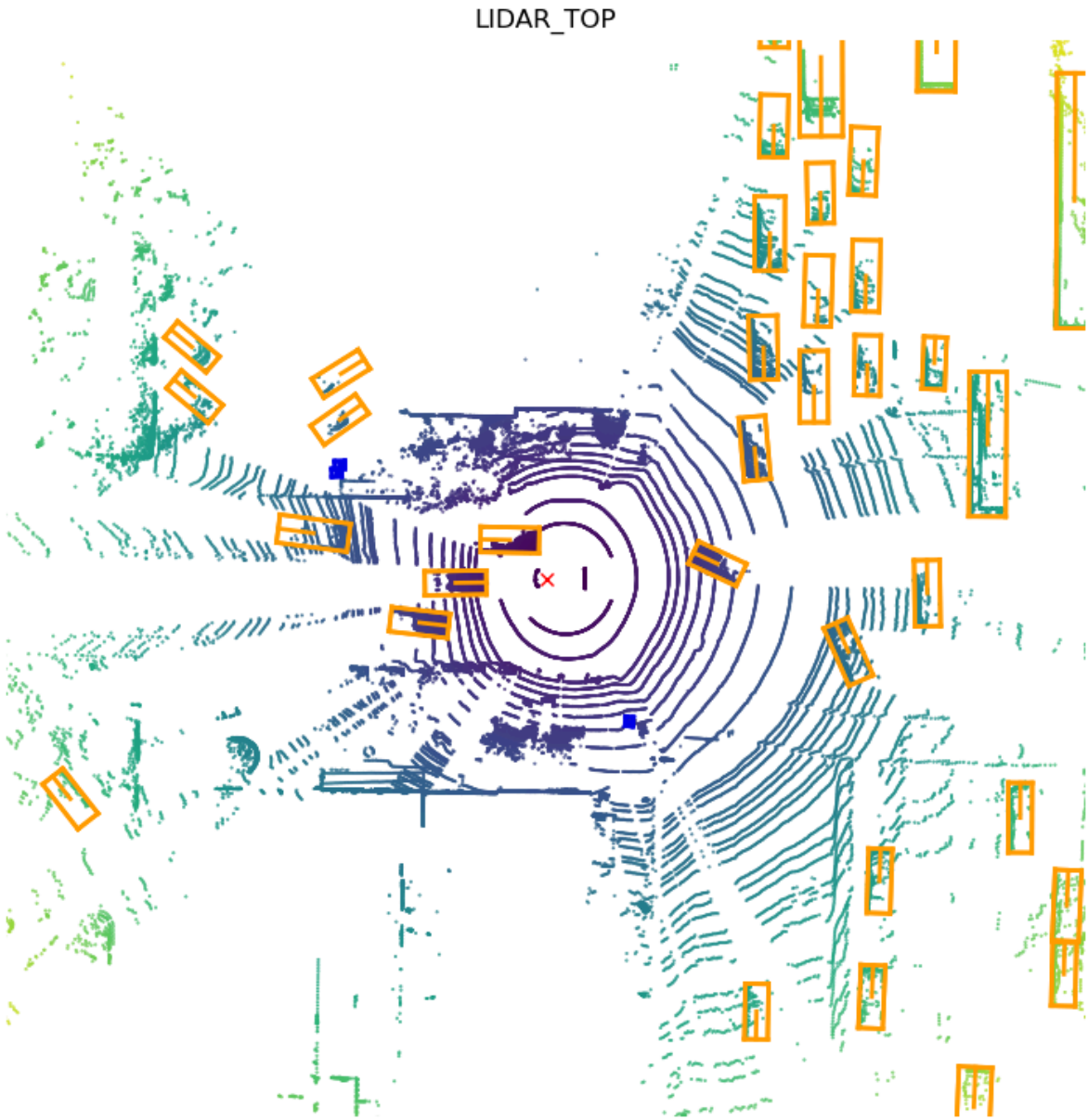
In [48]:
```
instance_token = my_instance['token']
lyft_dataset.render_instance(instance_token)
```



In [49]:
```
lyft_dataset.render_annotation(my_instance['last_annotation_token'])
```

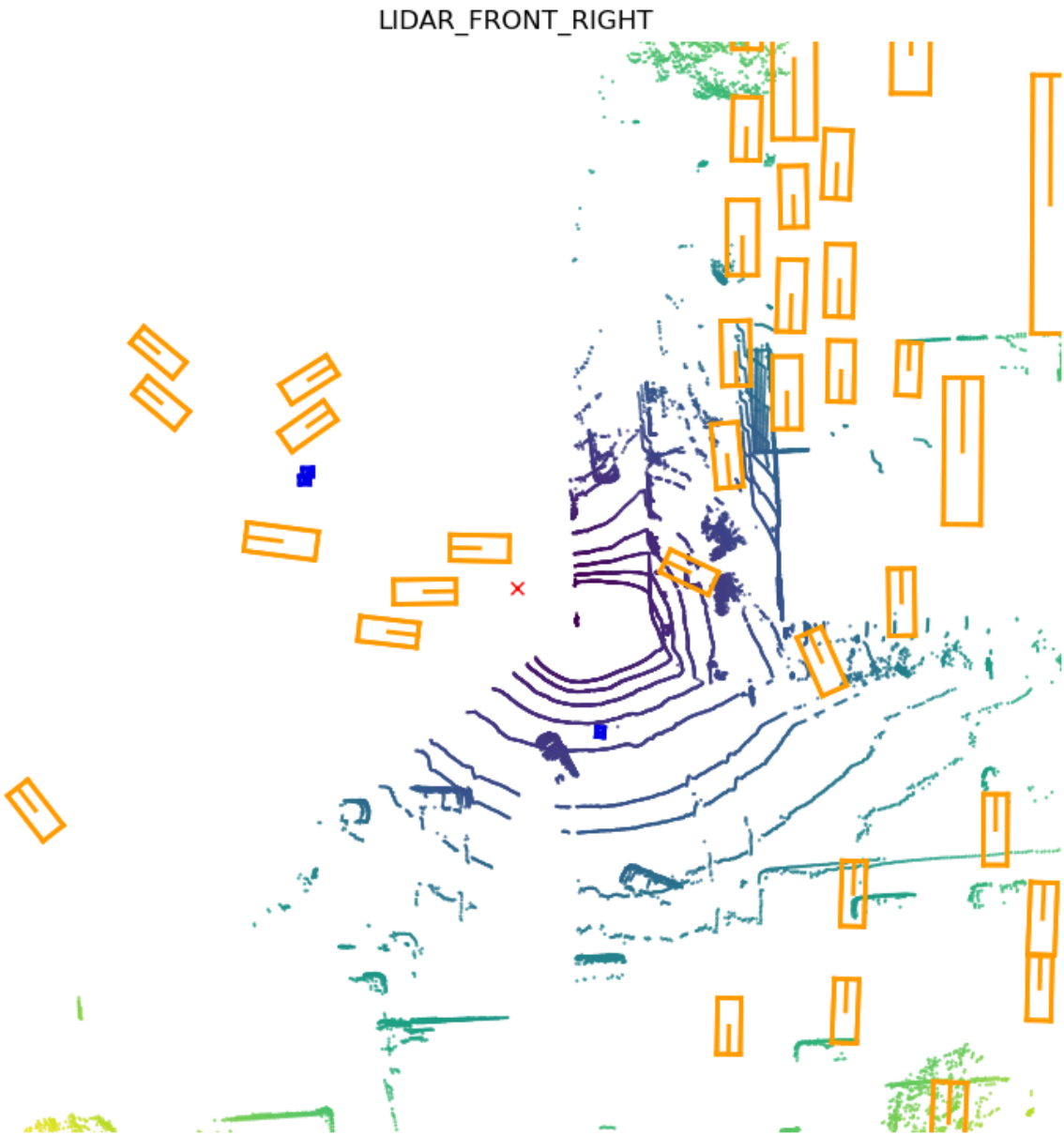


In [50]:
```
my_scene = lyft_dataset.scene[0]
my_sample_token = my_scene["first_sample_token"]
my_sample = lyft_dataset.get('sample', my_sample_token)
lyft_dataset.render_sample_data(my_sample['data']['LIDAR_TOP'], nsweeps=5)
```



In [51]:
```
my_scene = lyft_dataset.scene[0]
my_sample_token = my_scene["first_sample_token"]
my_sample = lyft_dataset.get('sample', my_sample_token)
lyft_dataset.render_sample_data(my_sample['data']['LIDAR_FRONT_LEFT'], nsweeps=5)
```

In [52]:
```
my_scene = lyft_dataset.scene[0]
my_sample_token = my_scene["first_sample_token"]
my_sample = lyft_dataset.get('sample', my_sample_token)
lyft_dataset.render_sample_data(my_sample['data']['LIDAR_FRONT_RIGHT'], nsweeps=5)
```



In [53]:

```
def generate_next_token(scene):
    scene = lyft_dataset.scene[scene]
    sample_token = scene['first_sample_token']
    sample_record = lyft_dataset.get("sample", sample_token)

    while sample_record['next']:
        sample_token = sample_record['next']
        sample_record = lyft_dataset.get("sample", sample_token)

        yield sample_token

def animate_images(scene, frames, pointsensor_channel='LIDAR_TOP', interval=1):
    cams = [
        'CAM_FRONT',
        'CAM_FRONT_RIGHT',
        'CAM_BACK_RIGHT',
        'CAM_BACK',
        'CAM_BACK_LEFT',
        'CAM_FRONT_LEFT',
    ]

    generator = generate_next_token(scene)

    fig, axs = plt.subplots(
        2, len(cams), figsize=(3*len(cams), 6),
```

```
        sharex=True, sharey=True, gridspec_kw = {'wspace': 0, 'hspace': 0.1}
    )

    plt.close(fig)

    def animate_fn(i):
        for _ in range(interval):
            sample_token = next(generator)

        for c, camera_channel in enumerate(cams):
            sample_record = lyft_dataset.get("sample", sample_token)

            pointsensor_token = sample_record["data"][pointsensor_channel]
            camera_token = sample_record["data"][camera_channel]

            axs[0, c].clear()
            axs[1, c].clear()

            lyft_dataset.render_sample_data(camera_token, with_anns=False, ax=axs[0
            lyft_dataset.render_sample_data(camera_token, with_anns=True, ax=axs[1,

            axs[0, c].set_title("")
            axs[1, c].set_title("")

    anim = animation.FuncAnimation(fig, animate_fn, frames=frames, interval=interva

    return anim
```

In [54]:
```
anim = animate_images(scene=3, frames=100, interval=1)
HTML(anim.to_jshtml(fps=8))
```

```
Animation size has reached 21402175 bytes, exceeding the limit of 20971520.0. If y
ou're sure you want a larger animation embedded, set the animation.embed_limit rc
parameter to a larger value (in MB). This and further frames will be dropped.
```

Out[54]:

Once Loop Reflect

In [55]:
```
anim = animate_images(scene=7, frames=50, interval=1)
HTML(anim.to_jshtml(fps=8))
```

```
Animation size has reached 21235543 bytes, exceeding the limit of 20971520.0. If y
ou're sure you want a larger animation embedded, set the animation.embed_limit rc
parameter to a larger value (in MB). This and further frames will be dropped.
```

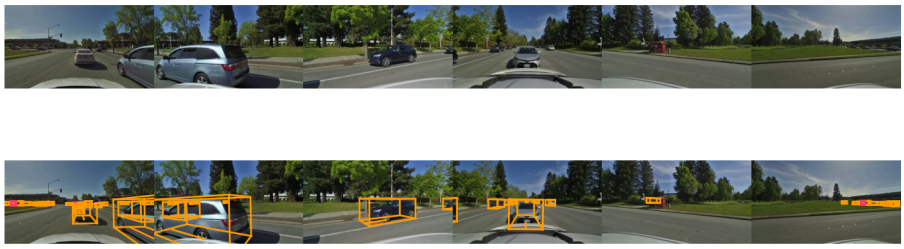
Out[55]:

Once Loop Reflect

In [56]:
```
anim = animate_images(scene=4, frames=50, interval=1)
HTML(anim.to_jshtml(fps=8))
```

```
Animation size has reached 21427285 bytes, exceeding the limit of 20971520.0. If y
ou're sure you want a larger animation embedded, set the animation.embed_limit rc
parameter to a larger value (in MB). This and further frames will be dropped.
```

Out[56]:



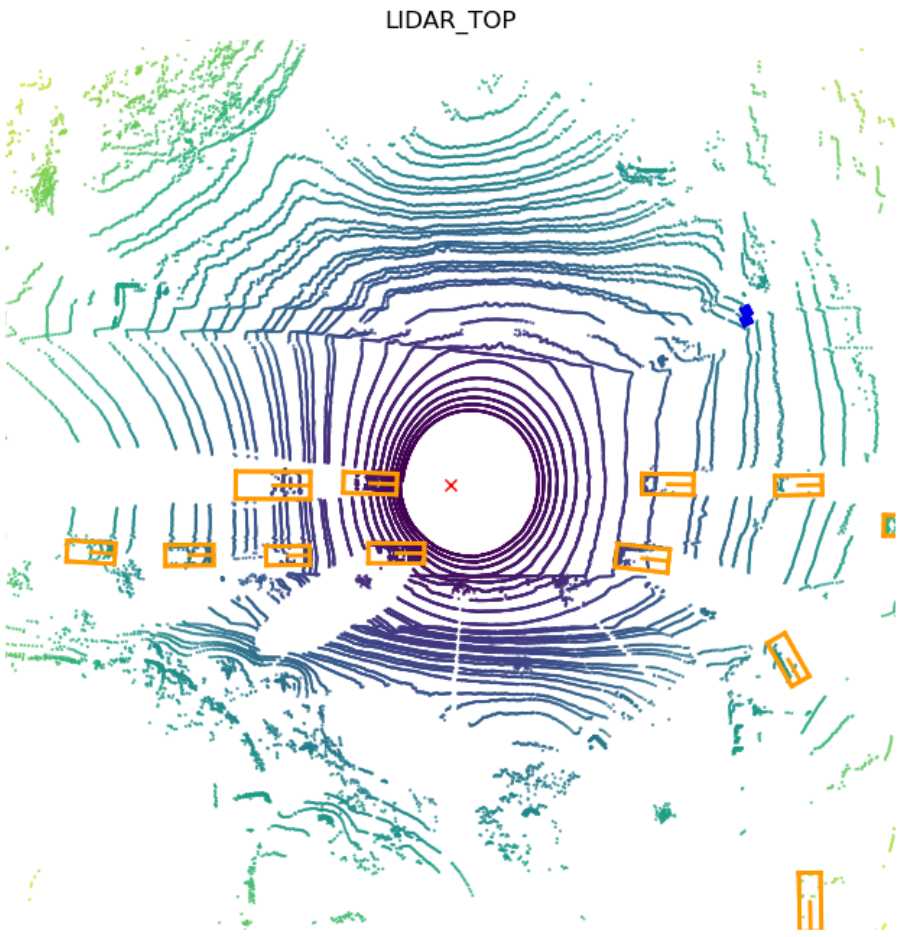
Once Loop Reflect

In [57]:
```
def animate_lidar(scene, frames, pointsensor_channel='LIDAR_TOP', with_anns=True, i
    generator = generate_next_token(scene)

    fig, axs = plt.subplots(1, 1, figsize=(8, 8))
    plt.close(fig)

    def animate_fn(i):
        for _ in range(interval):
            sample_token = next(generator)

        axs.clear()
        sample_record = lyft_dataset.get("sample", sample_token)
        pointsensor_token = sample_record["data"][pointsensor_channel]
        lyft_dataset.render_sample_data(pointsensor_token, with_anns=with_anns, ax=

    anim = animation.FuncAnimation(fig, animate_fn, frames=frames, interval=interva

    return anim
```

In [58]:
```
anim = animate_lidar(scene=5, frames=100, interval=1)
HTML(anim.to_jshtml(fps=8))
```

```
Animation size has reached 21094019 bytes, exceeding the limit of 20971520.0. If y
ou're sure you want a larger animation embedded, set the animation.embed_limit rc
parameter to a larger value (in MB). This and further frames will be dropped.
```

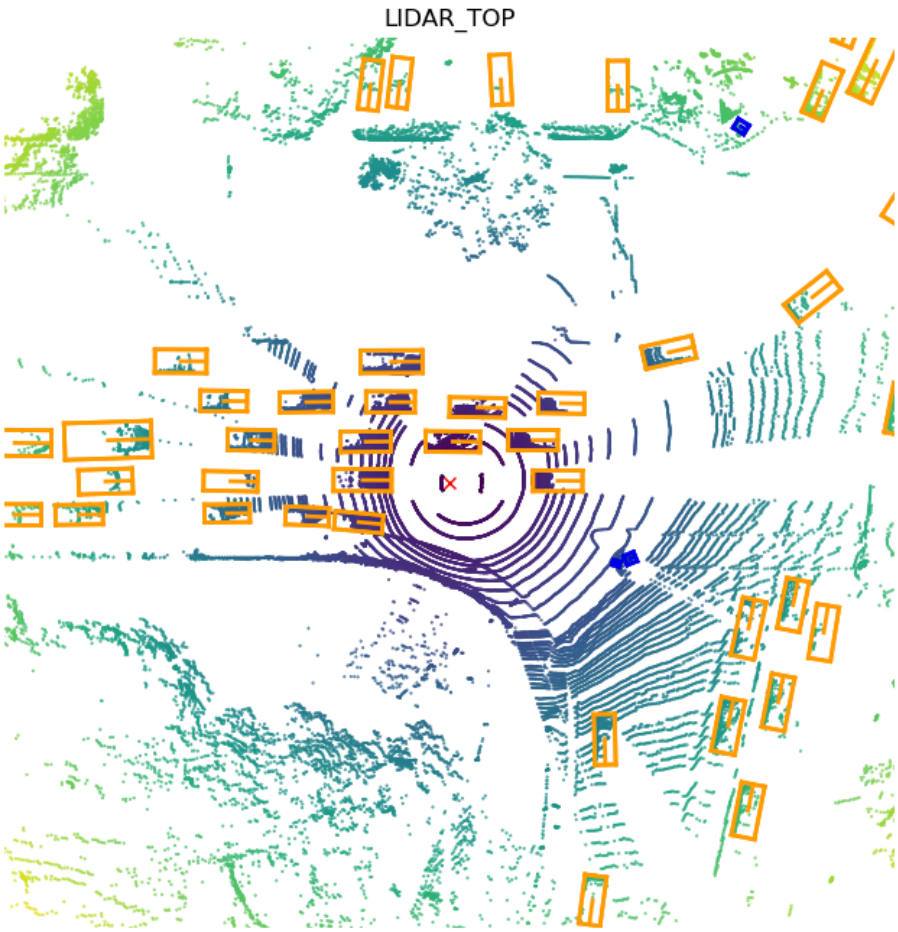
Out[58]:



Once Loop Reflect

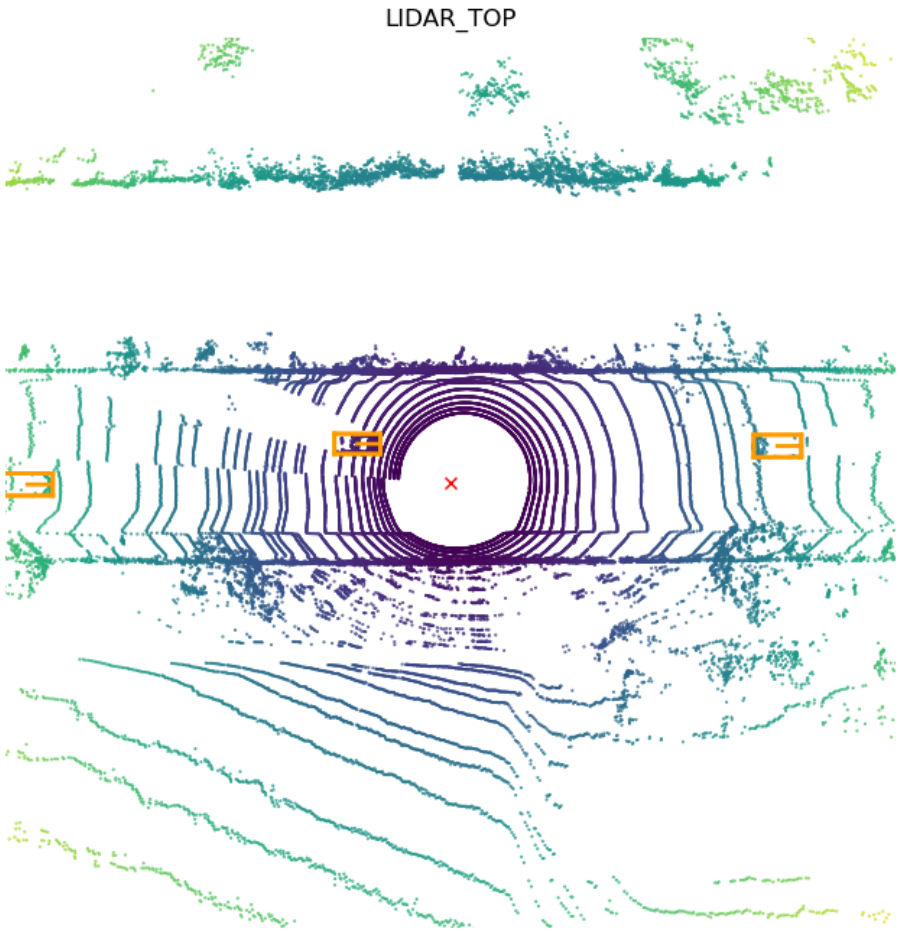
In [59]:
```
anim = animate_lidar(scene=25, frames=50, interval=1)
HTML(anim.to_jshtml(fps=8))
```

Out[59]:



In [60]:
```
anim = animate_lidar(scene=10, frames=50, interval=1)
HTML(anim.to_jshtml(fps=8))
```

Out[60]:



In [61]:

```python
from sklearn.metrics import precision_score, recall_score
def calculate_precision_recall(y_true, y_pred):
    """
    Calculate precision and recall for object detection.

    Parameters:
        y_true (numpy.ndarray): Ground truth labels (1 for object presence, 0 for b
        y_pred (numpy.ndarray): Predicted labels (1 for object presence, 0 for back

    Returns:
        precision (float): Precision score.
        recall (float): Recall score.
    """
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    return precision, recall

def calculate_iou(bbox_true, bbox_pred):
    """
    Calculate Intersection over Union (IoU) for bounding boxes.

    Parameters:
```

```
        bbox_true (tuple): Ground truth bounding box in format (x_min, y_min, x_max
        bbox_pred (tuple): Predicted bounding box in format (x_min, y_min, x_max, y

    Returns:
        iou (float): Intersection over Union score.
    """
    x_min_true, y_min_true, x_max_true, y_max_true = bbox_true
    x_min_pred, y_min_pred, x_max_pred, y_max_pred = bbox_pred

    x_left = max(x_min_true, x_min_pred)
    y_top = max(y_min_true, y_min_pred)
    x_right = min(x_max_true, x_max_pred)
    y_bottom = min(y_max_true, y_max_pred)

    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection_area = (x_right - x_left) * (y_bottom - y_top)

    true_area = (x_max_true - x_min_true) * (y_max_true - y_min_true)
    pred_area = (x_max_pred - x_min_pred) * (y_max_pred - y_min_pred)

    union_area = true_area + pred_area - intersection_area

    iou = intersection_area / union_area

    return iou

y_true = np.array([1, 0, 1, 1, 0])
y_pred = np.array([1, 1, 0, 1, 0])

# Calculate precision and recall
precision, recall = calculate_precision_recall(y_true, y_pred)
print("Precision:", precision)
print("Recall:", recall)

# Example bounding boxes for IoU calculation
bbox_true = (50, 50, 100, 100)
bbox_pred = (60, 60, 120, 120)

# Calculate IoU
iou = calculate_iou(bbox_true, bbox_pred)
print("IoU:", iou)
```

```
Precision: 0.6666666666666666
Recall: 0.6666666666666666
IoU: 0.35555555555555557
```

In [ ]: