Coverage for **install/environmental_model/lib/python3.8/site-packages/environmental_model/environmental_model.py** : 89%

157 statements | 139 run | 18 missing | 0 excluded

```python
"""
Sensor Fusion Node that fuses data from camera, LiDAR, and object detection.
"""

import rclpy
from rclpy.node import Node
from cv_bridge import CvBridge
from rclpy.qos import HistoryPolicy, QoSProfile, ReliabilityPolicy
import tf2_ros
import numpy as np
from sensor_msgs.msg import Image, LaserScan, CameraInfo
from geometry_msgs.msg import PoseStamped
from vision_msgs.msg import Detection2DArray
import cv2

# quality of service profile for the lidar scan subscription
QOS_PROFILE = QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT,
                         history=HistoryPolicy.KEEP_LAST, depth=1)

# mapping class IDs of retrained SSD model to the class labels
CLASS_MAP = {
    1: 'unlabeled',
    2: 'potted plant',
    3: 'shopping mall',
    4: 'marina',
    5: 'traffic light',
    6: 'car',
    7: 'person',
    8: 'wheelchair',
    9: 'shopping cart',
    10: 'mini',
    11: 'traffic cone',
    12: 'adapt car',
    13: 'truck',
    14: 'spotfinder car'
}

# Sensor Fusion Node
class SensorFusionNode(Node):
    '''class for sensor fusion node that fuses data from camera, LiDAR, and object detection'''

    def __init__(self):
        super().__init__('sensor_fusion_node')
        # Subscriptions
        self.subscription_camera = self.create_subscription(
            Image, '/camera/color/image_raw', self.image_callback, 10)
        self.subscription_detections = self.create_subscription(
            Detection2DArray, '/detectnet/detections', self.detection_callback, 10)
        self.subscription_lidar = self.create_subscription(
            LaserScan, '/scan', self.lidar_callback, QOS_PROFILE)
        self.subscription_pose = self.create_subscription(
            PoseStamped, '/ego_vehicle_pose', self.pose_callback, 10)
        self.subscription_camera_info = self.create_subscription(
            CameraInfo, '/camera/color/camera_info', self.camera_info_callback, 10)

        # Publishers
        self.image_pub = self.create_publisher(Image, '/camera/depth_annotated_images', 10)
        self.filtered_lidar_pub = self.create_publisher(LaserScan, '/filtered_scan', 10)

        # Transform listener
        self.tf_buffer = tf2_ros.Buffer()
        self.tf_listener = tf2_ros.TransformListener(self.tf_buffer, self)

        # Initialize variables
        self.current_lidar_points = []
        self.camera_intrinsics = None
        self.cv_bridge = CvBridge()
        self.current_image = None

        # Camera FOV in radians
        self.camera_fov = np.deg2rad(69.0)  # 69 degrees FOV

        # Lidar mounting angle offset in radians
        self.lidar_angle_offset = np.pi / 3  # 60 degrees

    # camera callback
    def image_callback(self, msg):
        '''Callback function for camera image subscription'''
        self.get_logger().info('Received image')
        self.current_image = self.cv_bridge.imgmsg_to_cv2(msg, "bgr8")

    # detection callback
    def detection_callback(self, msg):
        '''Callback function for object detection subscription'''
        self.get_logger().info('Received detections')
        if self.camera_intrinsics is None or self.current_image is None:
            return

        # Get camera intrinsic parameters
        f_x, f_y, c_x, c_y = self.camera_intrinsics
        # Get transform from camera_link to origin_laser_frame
        transform = self.get_transform('camera_link', 'origin_laser_frame')
        if transform is None:
            return

        # Project LiDAR points to image plane
        image_points = self.project_lidar_to_image(self.current_lidar_points,
                                                   transform, f_x, f_y, c_x, c_y)

        annotated_image = self.current_image.copy()

        # Loop through each detection
        for detection in msg.detections:
```

```
            self.get_logger().info(
                f'Detection BBox Center: '
                f'({detection.bbox.center.position.x}, {detection.bbox.center.position.y}), '
                f'Size: ({detection.bbox.size_x}, {detection.bbox.size_y})')
            # Calculate distance to the detected object
            distance, img_x, img_y = self.calculate_distance(detection, image_points)
            if distance:
                annotated_image = self.draw_bounding_box(annotated_image, detection,
                                                         distance, img_x, img_y)

        # Publish annotated image
        annotated_image_msg = self.cv_bridge.cv2_to_imgmsg(annotated_image, "bgr8")
        self.image_pub.publish(annotated_image_msg)

    # lidar callback
    def lidar_callback(self, msg):
        '''Callback function for LiDAR scan subscription'''
        self.get_logger().info('Received LiDAR scan')
        # Process LiDAR points
        filtered_points = self.process_lidar(msg)
        self.publish_filtered_lidar(msg)
        self.current_lidar_points = filtered_points

    # ego vehicle pose callback(for future use)
    def pose_callback(self, msg):
        '''Callback function for ego vehicle pose subscription'''
        pass

    # camera info callback
    def camera_info_callback(self, msg):
        '''Callback function for camera info subscription'''
        self.get_logger().info('Received Camera intrinsics')
        self.camera_intrinsics = (
            msg.k[0],  # f_x
            msg.k[4],  # f_y
            msg.k[2],  # c_x
            msg.k[5]   # c_y
        )
        self.get_logger().info(
            f'Camera intrinsics: f_x={msg.k[0]}, f_y={msg.k[4]}, c_x={msg.k[2]}, c_y={msg.k[5]}')

    # transform lookup helper function
    def get_transform(self, target_frame, source_frame):
        '''Lookup transform between target_frame and source_frame'''
        try:
            transform = self.tf_buffer.lookup_transform(target_frame, source_frame,
                                                        rclpy.time.Time())
            return transform
        except tf2_ros.LookupException:
            self.get_logger().info('Transform not found')
            return None

    # project lidar points to image plane helper function
    def project_lidar_to_image(self, lidar_points, transform, f_x, f_y, c_x, c_y):
        '''Project LiDAR points to the image plane using the camera intrinsics'''
        self.get_logger().info('Projecting LiDAR points to image plane')
        lidar_points_camera = self.transform_points(lidar_points, transform)
        image_points = []
        for point in lidar_points_camera:
            z_, x_, r_ = point  # z_ is the depth in the x axis(R axis), x_ is the distance in the y axis(G axis), r_ is the range of the point from the camera
            y_ = 0            # y_ is the distance in the z axis(B axis) (0 for 2D lidar)

            if r_ == 0:       # Avoid division by zero
                continue

            # Project 3D point to 2D image plane
            img_x = (x_ * f_x) / z_ + c_x
            img_y = (y_ * f_y) / z_ + c_y
            image_points.append((img_x, img_y, r_))
        return image_points

    # transform points helper function
    def transform_points(self, points, transform):
        '''Transform points using the given transform'''
        self.get_logger().info('Starting to transform points')
        transformed_points = []
        translation = transform.transform.translation

        for point in points:
            x_, y_, r_ = point
            # Apply translation
            transformed_point = np.array([x_ + translation.x, y_ + translation.y, r_])
            transformed_points.append(transformed_point)
        return transformed_points

    # calculate distance helper function
    def calculate_distance(self, detection, image_points):
        '''Calculate the distance to the detected object using
        the closest LiDAR point within the bounding box'''
        self.get_logger().info('Starting to calculate distance')
        # Calculate bounding box coordinates
        x_1 = detection.bbox.center.position.x - detection.bbox.size_x / 2
        x_2 = detection.bbox.center.position.x + detection.bbox.size_x / 2
        # Filter points within the bounding box
        distances = [(p[2], p[0], p[1]) for p in image_points if x_1 <= p[0] <= x_2]   # p[0] = img_x, p[1] = img_y, p[2] = r
        self.get_logger().info(f'Distances: {distances}')
        if distances:
            min_distance, img_x, img_y = min(distances)
            return min_distance, img_x, img_y
        return None, None, None

    # draw bounding box helper function
    def draw_bounding_box(self, image, detection, distance, img_x, img_y):
        '''Draw bounding box around the detected object and
        label it with the distance to the object'''
        self.get_logger().info('Starting to draw BBox')
        x_1 = int(detection.bbox.center.position.x - detection.bbox.size_x / 2)
        y_1 = int(detection.bbox.center.position.y - detection.bbox.size_y / 2)
        x_2 = int(detection.bbox.center.position.x + detection.bbox.size_x / 2)
        y_2 = int(detection.bbox.center.position.y + detection.bbox.size_y / 2)

```

```
        cv2.rectangle(image, (x_1, y_1), (x_2, y_2), (0, 255, 0), 2)
        if detection.results:
            class_id_ = (int(ord(detection.results[0].hypothesis.class_id))+1)
            label = f'{CLASS_MAP.get(class_id_, 0)}: {distance:.2f}m'
            cv2.putText(image, label, (x_1, y_1 - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
            self.get_logger().info(f'Drawn bounding box: {label}')
            if img_x is not None and img_y is not None:
                # Draw the closest LiDAR point
                cv2.circle(image, (int(img_x), int(img_y)), 5, (255, 0, 0), -1)
                self.get_logger().info(f'Drawn LiDAR point: ({img_x}, {img_y})')
        else:
            self.get_logger().warn('Detection results are empty, skipping label')
        return image

    # lidar points processing helper function
    def process_lidar(self, scan):
        '''Process LiDAR scan and return points within the camera FOV and valid range'''
        self.get_logger().info('Processing laserscan points')
        points = []
        angle = scan.angle_min
        # Loop through each point in the scan
        for polar_r in scan.ranges:
            # Adjust angle for lidar mounting angle offset
            adjusted_angle = angle + self.lidar_angle_offset
            # Check if the point is within the camera FOV
            if -self.camera_fov / 2 <= adjusted_angle <= self.camera_fov / 2:
                # Check if the point is within the valid range
                if scan.range_min < polar_r < scan.range_max:
                    # convert polar coordinates to cartesian coordinates
                    cartesian_x = polar_r * np.cos(adjusted_angle)
                    cartesian_y = polar_r * np.sin(adjusted_angle)
                    points.append((cartesian_x, cartesian_y, polar_r))  # Store r as the third element instead of z
            angle += scan.angle_increment
        return points

    # publish filtered lidar for visualization
    def publish_filtered_lidar(self, original_scan):
        '''Publish filtered LiDAR scan for visualization'''
        filtered_scan = LaserScan()
        filtered_scan.header = original_scan.header
        filtered_scan.angle_min = original_scan.angle_min
        filtered_scan.angle_max = original_scan.angle_max
        filtered_scan.angle_increment = original_scan.angle_increment
        filtered_scan.time_increment = original_scan.time_increment
        filtered_scan.scan_time = original_scan.scan_time
        filtered_scan.range_min = original_scan.range_min
        filtered_scan.range_max = original_scan.range_max

        filtered_scan.ranges = [0.0] * len(original_scan.ranges)
        angle = original_scan.angle_min

        for i, r_ in enumerate(original_scan.ranges):
            adjusted_angle = angle + self.lidar_angle_offset
            if original_scan.range_min < r_ < original_scan.range_max:
                if -self.camera_fov / 2 <= adjusted_angle <= self.camera_fov / 2:
                    filtered_scan.ranges[i] = r_
            angle += original_scan.angle_increment

        self.filtered_lidar_pub.publish(filtered_scan)


def main(args=None):
    rclpy.init(args=args)
    node = SensorFusionNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

« index   coverage.py v4.5.2, created at 2024-06-26 19:25