Brendan Neal 1001160236|nealbre1 2018-11-05

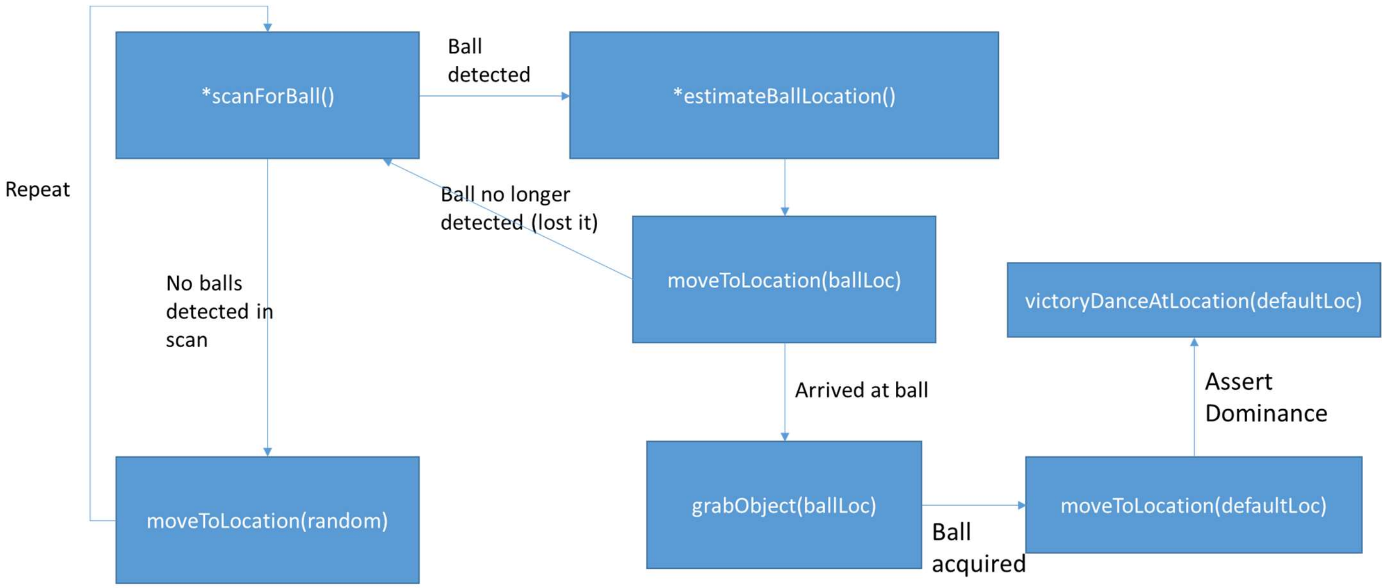
CSC420 Assignment 4 Question 1

Part A: Brainstorming

- What will the robot encounter?
  - People (likely tennis players)
  - Tennis balls
  - Tennis net
  - Walls of tennis court
  - Floor of tennis court
- Training Data
  - Object Classification / Detection
    - Self-driving-car-esque training data (inputs are raw images, outputs are segmentation masks or bounding boxes)
    - Classes: everything mentioned above (“what will the robot encounter?”)
- Robot World Model
  - Collection of (class, location) pairs (keep track of where everything is)
  - Robot’s current 3D location
  - Size of tennis court (robot shouldn’t try to exit the court / crash into walls)
  - Robot’s default 3D location
  - 3D location of net (so robot can get around it)
- Robot Challenges
  - Don’t run into anything (walls, people, tennis net, etc.)
  - Avoid moving objects (e.g. people moving around while robot is getting ball)
  - Ball may be out of line of sight from default location
  - Robot needs to find a way around the net

Assumption: the moveToLocation() method automatically avoids collisions. This method would have to use the model of the robot’s world to plot paths around objects that the robot is likely to collide with.

Part B: Flow Chart



Part C: Pseudocode

```
detectObjects(img):
	Runs a trained object detection neural network on img to detect objects.
	Possible classes:
		People
		Balls
		Tennis Net
		Court Walls
		Court Floor
scanForBall():
	for each slice of 360 degree rotation:
		faceDirection(slice)
		img = takePicture()
		detections = detectObjects(img)
		for det in detections:
			if det.class is TennisBall:
				return img
	return None

estimateBallLocation(i1, i2, detections):
	focal_length, base_width = cameraParams
	disparity = patchMatchDisparity(i1, i2)
	z = focal_length * base_width / disparity
	x = x_coords * z / focal_length
	y = y_coords * z / focal_length

	ball_detection = (find ball detection in detections)

	# Use similar method to question 2D to find center of mass
	ball_location = center of mass of ball points using ball_detection, z, x, y

	return ball_location
```

Part D: Robot Eye Loss (BONUS)

Losing an eye/camera means that the robot can no longer use triangulation of two simultaneous images to calculate the 3D locations of objects. However, this can be compensated for using the following method:

1) Capture image I_1
2) Move ϵ meters to the left or right (in x direction)
3) Capture image I_2
4) Run triangulation between I_1 and I_2 using a base width of ϵ meters

This approach effectively captures two images from different perspectives and therefore partially emulates the simultaneous capture of 2 images by dual cameras. This method is very similar to how birds with non-stereo vision use head movement to capture multiple views to emulate depth perception. However, there are a few failure cases for this method:

- If the robot is unable to move a sufficient distance horizontally, then the effective base width will be very small or close to zero. If the base width is small enough, the relative error in the base width will be quite large. Since depth (z) is inversely proportional to base width, a large relative error from a small base width will result in large uncertainties in estimated locations.

- If the two images are very different (e.g. a person walks in front of the robot when it is capturing I_2), matching points to create a disparity image will fail. If a disparity image cannot be created properly, then triangulation cannot be used to estimate 3D locations properly.

- This method performs poorly if the robot has to continually estimate 3D locations while it is moving. Essentially, the method will force the robot to obey a zig-zag like path when moving so it can capture the two perspectives needed to estimate 3D locations. Obviously, such a path is inefficient and the robot will perform its job slower as a result.

- If objects are moving around faster than the robot can capture its pseudo-stereo images, the robot will be unable to accurately estimate their real locations. The robot will be more likely to crash into things if it cannot estimate object locations.

CSC420 Assignment 4 Question 2

Brendan Neal | 1001160236 | nealbre1

Imports and some helper functions

In [1]:
```
import numpy as np
from scipy import spatial, cluster
import cv2 as cv
import math
import re
import os
import tensorflow as tf
from collections import Counter

from matplotlib import pyplot as plot

# Make the plot a certain size
plot.rcParams["figure.figsize"] = [8, 6]


def display_image(img, file_name=None, save_norm=True, save_type=np.uint8):
    """
    Shows an image (max-min normalized to 0-255), and saves it if a filename is given
    save_norm = whether to save the normalized image
    save_type = what datatype to save the image as
    """

    flt_img = img.astype(float)
    img_max, img_min = np.max(flt_img), np.min(flt_img)
    norm = (((flt_img - img_min) / (img_max - img_min)) * 255).astype(np.uint8)

    if len(img.shape) == 2:
        plot.imshow(norm, cmap='gray')
    elif (len(img.shape) == 3):
        plot.imshow(cv.cvtColor(norm, cv.COLOR_BGR2RGB))
    plot.show()

    to_save = norm if save_norm else flt_img
    if file_name:
        cv.imwrite(file_name, to_save)
```

Question 2a: Depth Calculation

Formula for depth calculation:

$$z = \frac{f \times b}{d}$$

Where:

z : z-coordinate/depth of the pixel (meters)

f : focal length of the camera(s) (pixels)

b : base width of the two camera (meters)

d : x-disparity of the pixel (pixels)

Code to grab calculate depth image and get camera params:

In [2]:

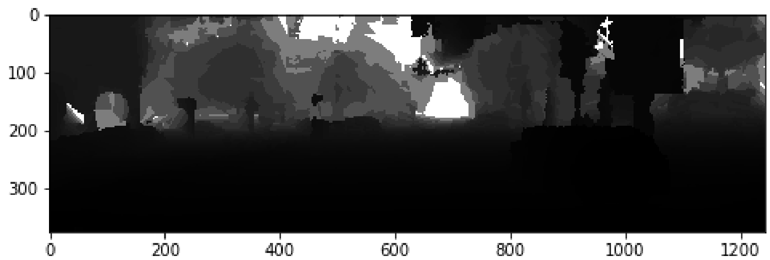
```
def depth_image(disparity_image, base_width, focal_length):
    """
    Creates a depth image using a disparity image and
    some camera parameters (base width, focal length)
    """
    numerator = base_width * focal_length

    # Avoid div 0 errors
    denominator = disparity_image.astype(float)
    denominator[denominator == 0] = 1
    result = numerator / denominator

    # Set what should have been infinity to maximum
    result[denominator == 0] = np.max(result)

    return result


def get_camera_params(param_path):
    """
    Returns dictionary of camera parameter values
    stored in the file at param_path
    """

    # Lines of the file are of form (param : value)
    line_regex = re.compile(r"(?P<param>(\w+)):(\s+)(?P<value>(\d+(\.\d+)?))")

    with open(param_path, "r") as file:

        # Matching the regex to the lines in the file
        matches = (line_regex.match(line) for line in file.readlines())

        # Organize the parameters into a dictionary for easy access
        return {
            match.group("param") : float(match.group("value"))
            for match in matches
        }
```

Running the above code for the first 3 images in the *test* folder:

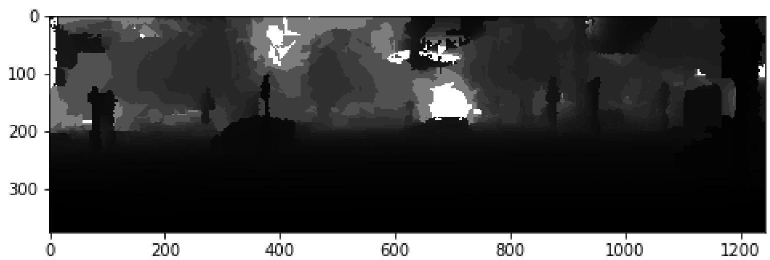
In [3]:

```
# Only need 3 image IDs in test
test_image_ids = [
    x.strip()
    for x in open("data/test/test.txt", "r").readlines()
]

for img_id in test_image_ids[:3]:
    camera_params = get_camera_params("data/test/calib/{}_allcalib.txt".format(img_id))
    disparity_img = cv.imread("data/test/results/{}_left_disparity.png".format(img_id), cv.IMREAD_GRAY
SCALE)
    depth_img = depth_image(disparity_img, camera_params["baseline"], camera_params["f"])

    print(img_id)
    display_image(depth_img, "q2a-{}-depth.png".format(img_id), save_norm=False, save_type=np.float32)
```

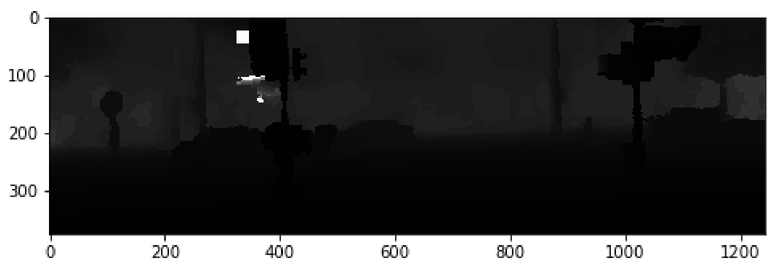
```
004945
```



```
004964
```



```
005002
```



Question 2b: Object Detection

Code to import and run a pre-trained tensorflow model:

```
In [4]: def load_frozen_graph(model_folder):
    """
    Adapted from:
    https://blog.metaflow.fr/tensorflow-how-to-freeze-a-model-and-serve-it-with-a-python-api-d4f3596b3
adc
    """
    # Load the graph definition
    frozen_graph_path = os.path.join(model_folder, r"frozen_inference_graph.pb")

    with tf.gfile.GFile(frozen_graph_path, "rb") as graph_file:
        graph_def = tf.GraphDef()
        graph_def.ParseFromString(graph_file.read())

    # Import the definition into a graph, return it
    with tf.Graph().as_default() as graph:
        tf.import_graph_def(graph_def, name="")

    # Return the loaded graph
    return graph

def detect_objects(graph, image, threshold):
    """
    Adapted from:
    https://github.com/tensorflow/models/blob/master/research/object_detection/object_detection_tutori
al.ipynb

    Changed to return numpy array of form:
    [[x_left, y_top, x_right, y_bottom, confidence, class]]

    Where each row represents a bounding box's diagonal coordinates,
    confidence score and classification.
    """
    with graph.as_default():
        with tf.Session() as sess:
            # Get handles to input and output tensors
            ops = tf.get_default_graph().get_operations()
            all_tensor_names = {output.name for op in ops for output in op.outputs}
            tensor_dict = {}

            for key in ['num_detections', 'detection_boxes', 'detection_scores',
                        'detection_classes', 'detection_masks']:
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                    tensor_dict[key] = tf.get_default_graph().get_tensor_by_name(tensor_name)

            if 'detection_masks' in tensor_dict:
                # The following processing is only for single image
                detection_boxes = tf.squeeze(tensor_dict['detection_boxes'], [0])
                detection_masks = tf.squeeze(tensor_dict['detection_masks'], [0])
                # Reframe is required to translate mask from box coordinates to image coordinates and
 fit the image size.
                real_num_detection = tf.cast(tensor_dict['num_detections'][0], tf.int32)
                detection_boxes = tf.slice(detection_boxes, [0, 0], [real_num_detection, -1])
                detection_masks = tf.slice(detection_masks, [0, 0, 0], [real_num_detection, -1, -1])
                detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
                    detection_masks, detection_boxes, image.shape[0], image.shape[1])
                detection_masks_reframed = tf.cast(
                    tf.greater(detection_masks_reframed, 0.5), tf.uint8)
                # Follow the convention by adding back the batch dimension
                tensor_dict['detection_masks'] = tf.expand_dims(
                    detection_masks_reframed, 0)

            image_tensor = tf.get_default_graph().get_tensor_by_name('image_tensor:0')

            # Run inference
            output = sess.run(tensor_dict,
                              feed_dict={image_tensor: np.expand_dims(image, 0)})

            ###### MY OWN CODE TO CHANGE FORMAT OF OUTPUT #######
            img_height, img_width = image.shape[:2]
            num_detections = int(output["num_detections"][0])
```

```
            # Return a numpy array where each row has the following form:
            #     (y_top, x_left, y_bottom, x_right, confidence, class)
            # Coordinates are NOT normalized (they are actual image coordinates)
            classes = output["detection_classes"][0][:num_detections]
            box_coords = output["detection_boxes"][0][:num_detections] * np.array([img_height, img_wid
th, img_height, img_width])
            scores = output["detection_scores"][0][:num_detections]

            # Stack the above matrices horizontally
            stacked = np.hstack((box_coords.astype(int),
                                 scores.reshape(-1, 1),
                                 classes.astype(int).reshape(-1, 1)))

            # Only take the rows where scores are above threshold
            return stacked[scores >= threshold]
```

Running code on images (using Mobile Net trained on the COCO dataset).

I am using a threshold of 0.3. It eliminates some false positives while still maintaining actual positive identifications. Unfortunately, high-confidence false positives still remain.

In [13]:
```
coco_mobile_net = load_frozen_graph("coco-mobile-net")

for img_id in test_image_ids:
    img = cv.imread("data/test/left/{}.jpg".format(img_id))
    detections = detect_objects(coco_mobile_net, img, threshold=0.3)

    # Save detections in test results folder
    np.save("data/test/results/{}_detections.npy".format(img_id), detections)
```

Question 2c: Display First 3 Image Visualizations

In [14]:
```
PERSON, BIKE, CAR, TRAFFIC_LIGHT = 1, 2, 3, 10

cls_to_name = {
    PERSON : "PERSON",
    BIKE : "BIKE",
    CAR : "CAR",
    TRAFFIC_LIGHT : "TRAFFIC LIGHT"
}

cls_to_col = {
    PERSON  : (255, 0, 0), # Blue
    BIKE  : (0, 255, 0), # Green
    CAR  : (0, 0, 255), # Red
    TRAFFIC_LIGHT : (255, 255, 0)# Cyan
}

for img_id in test_image_ids[:3]:
    detections = np.load("data/test/results/{}_detections.npy".format(img_id))
    img = cv.imread("data/test/left/{}.jpg".format(img_id))

    # Draw rectangles of detection on image
    for (y_top, x_left, y_bottom, x_right, confidence, cls) in detections:
        if cls in cls_to_name:
            top_lft = (int(x_left), int(y_top))
            bot_rgt = (int(x_right), int(y_bottom))


            cv.rectangle(img, top_lft, bot_rgt, cls_to_col[cls], 3)
            label_txt = cls_to_name[cls]
            label_coords = top_lft
            cv.putText(img, label_txt, label_coords, cv.FONT_HERSHEY_SIMPLEX, 0.65, cls_to_col[cls], 2
)

    # Show the image
    print(img_id)
    display_image(img, "q2c-{}-detections.png".format(img_id))
```

004945

004964

005002

Question 2d: Center of Mass Computation

Assuming points with equal mass, the center of mass of a collection of points will be the average point. However, note that within a detection box there are two distinct types of pixels: object and non-object pixels. If the object within the box is assumed to be the only object of significance, then the distribution of points will be bimodal between the object and the background pixels. Therefore, k-means clustering with k = 2 can be used to approximate the two means.

The following formulas will be used to calculate each point (x, y, z):

$$z = \frac{f \times b}{d}$$
$$x = \frac{x_L \times z}{f}$$
$$y = \frac{y_L \times z}{f}$$

Where:

z : z-coordinate of the pixel (meters)

x : x-coordinate of the pixel (meters)

y : y-coordinate of the pixel (meters)

f : focal length of the camera(s) (pixels)

b : base width of the two camera (meters)

d : x-disparity of the pixel (pixels)

x_L : x-coordinate of the pixel in the left image (pixels)

y_L : y-coordinate of the pixel in the left (or right, they're assumed to be the same) image (pixels)

In [7]:
```python
def image_3d_coords(disparity, base_width, focal_length, center_x, center_y):
    """
    Produces an image where each pixel is of form [x, y, z]
    """

    # Compute Z/depth first
    z = depth_image(disparity, base_width, focal_length)

    # Shift raw coordinates by optical center
    raw_y, raw_x = np.indices(z.shape[:2])
    shifted_y, shifted_x = raw_y - center_y, raw_x - center_x

    # d = xL - xR --> xL = d + xR (assuming x-coordinates of disparity are xR)
    # y is assumed to be the same across both images
    yL = shifted_y
    xL = shifted_x + disparity

    x = (xL * z) / focal_length
    y = (yL * z) / focal_length

    return np.dstack((x, y, z))

def find_closest_mean(vectors, k):
    """
    Returns the mean vector with the smallest magnitude.
    (vectors is assumed to be a k-modal distribution)
    """

    # Run K-means
    centroids, labels = cluster.vq.kmeans2(vectors, k)

    # Determine magnitude of centroids
    centroid_mags = (centroids ** 2).sum(axis=1) ** 0.5
    closest_label = np.argmin(centroid_mags)

    # Return the closest centroid
    return centroids[closest_label]

def centers_of_mass(coords_3d, detections):
    """
    Takes an image of coordinates (each pixel is of form [x, y, z]),
    and an array of detections each of form
    [y_top, x_left, y_bottom, x_right, confidence, class]

    Computes a numpy array of form [[x, y, z]], where the i-th element
    is equal to the center of mass of the i-th detection's points
    """

    # Slices of given coordinate matrix (for each detection box)
    slice_inds = (
        row[:4].astype(int)
        for row in detections
    )
    slices = (
        coords_3d[y_top:y_bottom, x_left:x_right]
        for (y_top, x_left, y_bottom, x_right) in slice_inds
    )

    # Flattened (x, y) slices
    flat_slices = (
        s.reshape(np.product(s.shape[:2]), 3)
        for s in slices
    )

    # Return closest mean of the bimodal distribution
    return np.array([
        find_closest_mean(vecs, 2)
        for vecs in flat_slices
    ])
```

Running the above code on each image (saving computed centers of mass and coordinates in results)

In [8]:
```
for img_id in test_image_ids[:3]:

    # Compute 3D coordinates of pixels
    camera_params = get_camera_params("data/test/calib/{}_allcalib.txt".format(img_id))
    disparity = cv.imread("data/test/results/{}_left_disparity.png".format(img_id), cv.IMREAD_GRAYSCAL
E)

    baseline, f, px, py = [camera_params[k] for k in ["baseline", "f", "px", "py"]]
    coords = image_3d_coords(disparity, baseline, f, px, py)

    # Detections of image
    detections = np.load("data/test/results/{}_detections.npy".format(img_id))

    # Centers of mass for each detection, in order of detections
    mass_centers = centers_of_mass(coords, detections)

    # Save the centers of mass and coordinates for each image
    np.save("data/test/results/{}_3d_coords.npy".format(img_id), coords)
    np.save("data/test/results/{}_mass_centers.npy".format(img_id), mass_centers)
```

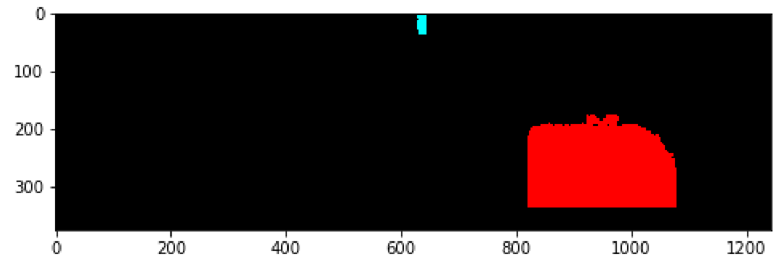
```
C:\ProgramData\Anaconda3\lib\site-packages\scipy\cluster\vq.py:660: UserWarning: One of the clusters
is empty. Re-run kmean with a different initialization.
  warnings.warn("One of the clusters is empty. "
```

Question 2e: Segmentation Using 3D Coordinates and Centers of Mass

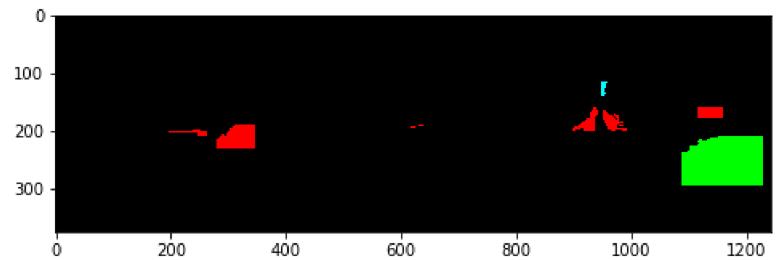
In [9]:
```
def segment_image(coords, detections, centers, max_dist):

    # The image to return
    ret = np.zeros(coords.shape, int)

    # Go through each detection
    for center, (y_top, x_left, y_bottom, x_right, confidence, cls) in zip(centers, detections):

        # Int-ify each bounding box coordinate for slicing
        yT, yB, xL, xR = int(y_top), int(y_bottom), int(x_left), int(x_right)

        # Get slice of coordinates and the center of mass
        coord_slice = coords[yT:yB, xL:xR]

        # Compute euclidean distance from center for each coordinate
        center_dists = ((coord_slice - center) ** 2).sum(axis=2) ** 0.5

        # Set the points within max_dist to the class color
        ret[yT:yB, xL:xR][center_dists < max_dist] = cls_to_col[cls]

    return ret
```

```
In [10]: for img_id in test_image_ids[:3]:

    coords = np.load("data/test/results/{}_3d_coords.npy".format(img_id))
    centers = np.load("data/test/results/{}_mass_centers.npy".format(img_id))
    detections = np.load("data/test/results/{}_detections.npy".format(img_id))

    # Segment the image with maximum center-distance of 3 meters
    segmented = segment_image(coords, detections, centers, 3)

    print(img_id)
    display_image(segmented, "q2e-{}-segmentation.png".format(img_id))
```

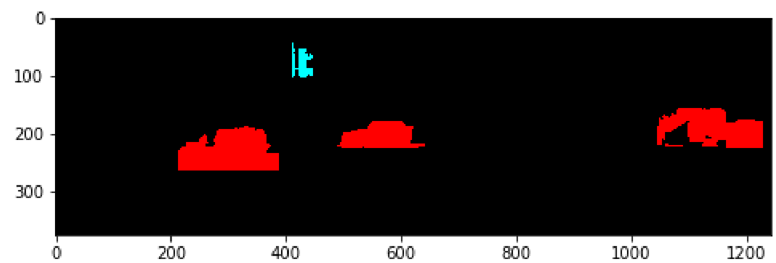
```
004945
```




```
004964
```




```
005002
```

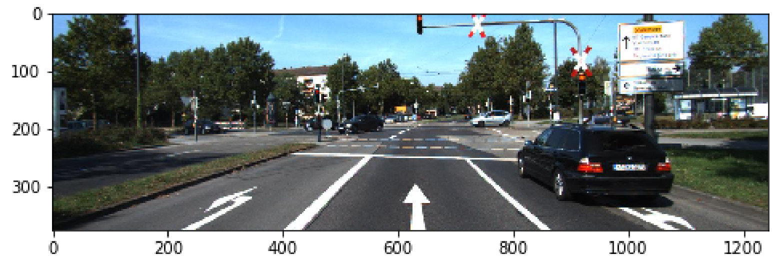


Question 2f: Textual Description Generation

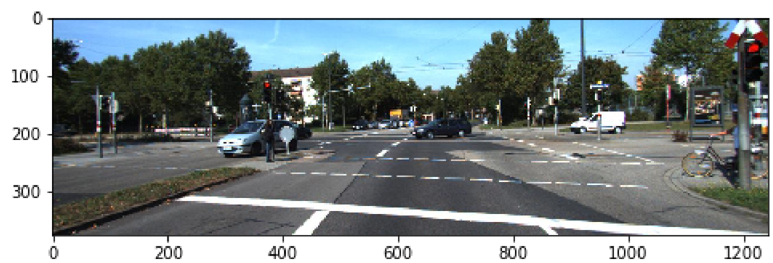
In [11]:
```python
def make_description(detections, centers):
    class_counts = Counter(det[-1] for det in detections)

    # Determine distance of centroids from camera, then the closest center
    center_dists = (centers ** 2).sum(axis=1) ** 0.5
    closest_center_ind = np.argmin(center_dists)

    # Determine closest object's detection information
    closest_center = centers[closest_center_ind]
    closest_dist = center_dists[closest_center_ind]
    closest_detection = detections[closest_center_ind]

    return "\n".join([

        # Describe counts of classes
        "Number of Cars: {}".format(class_counts[CAR]),
        "Number of People: {}".format(class_counts[PERSON]),
        "Number of Cyclists: {}".format(class_counts[BIKE]),

        # Whether or not image has traffic light
        "Nearby Traffic Light: {}".format("Yes" if class_counts[TRAFFIC_LIGHT] else "No"),

        # Describe the closest detection
        "There is a {cls} {dst}m away - {horz_amt}m {horz_dir}, {vert_amt}m {vert_dir}, {depth_amt}m b
ack".format(

            # Name of closest detection's class
            cls=cls_to_name[closest_detection[-1]].lower(),

            # Distance to closest detection
            dst=round(center_dists[closest_center_ind], 2),

            # Describing horizontal (x) displacement
            horz_amt = abs(round(closest_center[0], 2)),
            horz_dir = "right" if closest_center[0] > 0 else "left",

            # Describing vertical (y) displacement -
            # reverse because image coordinates are upside down
            vert_amt = abs(round(closest_center[1], 2)),
            vert_dir = "down" if closest_center[1] > 0 else "up",

            # Describing depth (z) displacement
            depth_amt = abs(round(closest_center[2], 2))
        )
    ])
```

In [12]:
```python
for img_id in test_image_ids[:3]:

    img = cv.imread("data/test/left/{}.jpg".format(img_id))
    detections = np.load("data/test/results/{}_detections.npy".format(img_id))
    centers = np.load("data/test/results/{}_mass_centers.npy".format(img_id))

    print(img_id)
    display_image(img)
    print(make_description(detections, centers))
    print("\n")
```

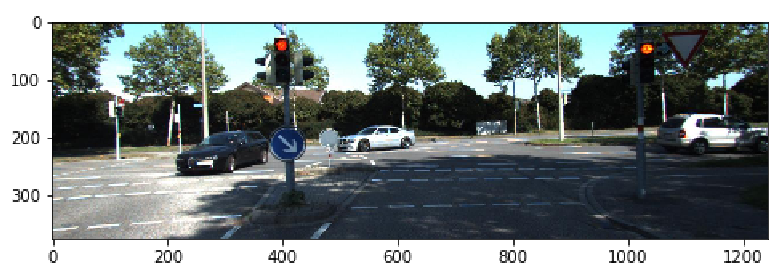
```
004945
```




```
Number of Cars: 4
Number of People: 0
Number of Cyclists: 0
Nearby Traffic Light: Yes
There is a car 9.09m away - 4.23m right, 0.96m down, 7.98m back


004964
```




```
Number of Cars: 6
Number of People: 0
Number of Cyclists: 1
Nearby Traffic Light: Yes
There is a bike 13.41m away - 8.44m right, 1.13m down, 10.36m back


005002
```




```
Number of Cars: 3
Number of People: 0
Number of Cyclists: 0
Nearby Traffic Light: Yes
There is a traffic light 7.98m away - 1.44m left, 1.03m up, 7.78m back
```