



AUTONOMOUS DRIVING

PROJECT

UNIVERSITY OF SALERNO, DIEM

PROJECTS SCENARIOS

Possible Scenarios

Lane merging.

Lane changing.

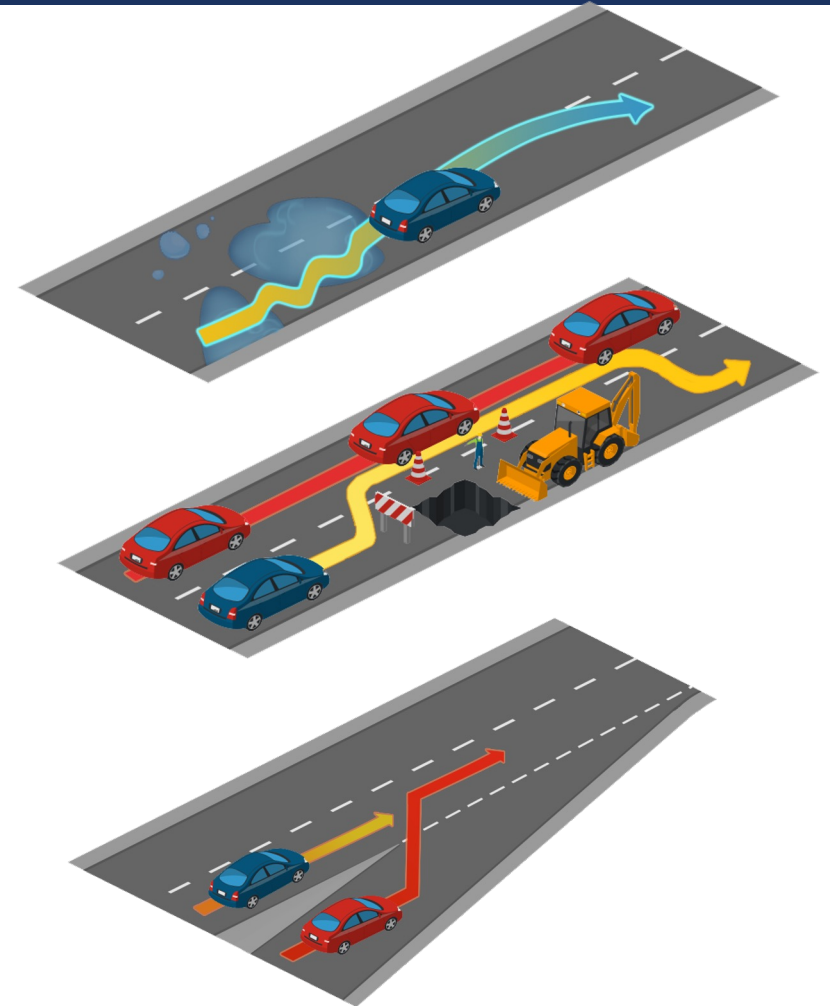
Negotiations at traffic intersections.

Negotiations at roundabouts.

Handling traffic lights and traffic signs.

Yielding to emergency vehicles.

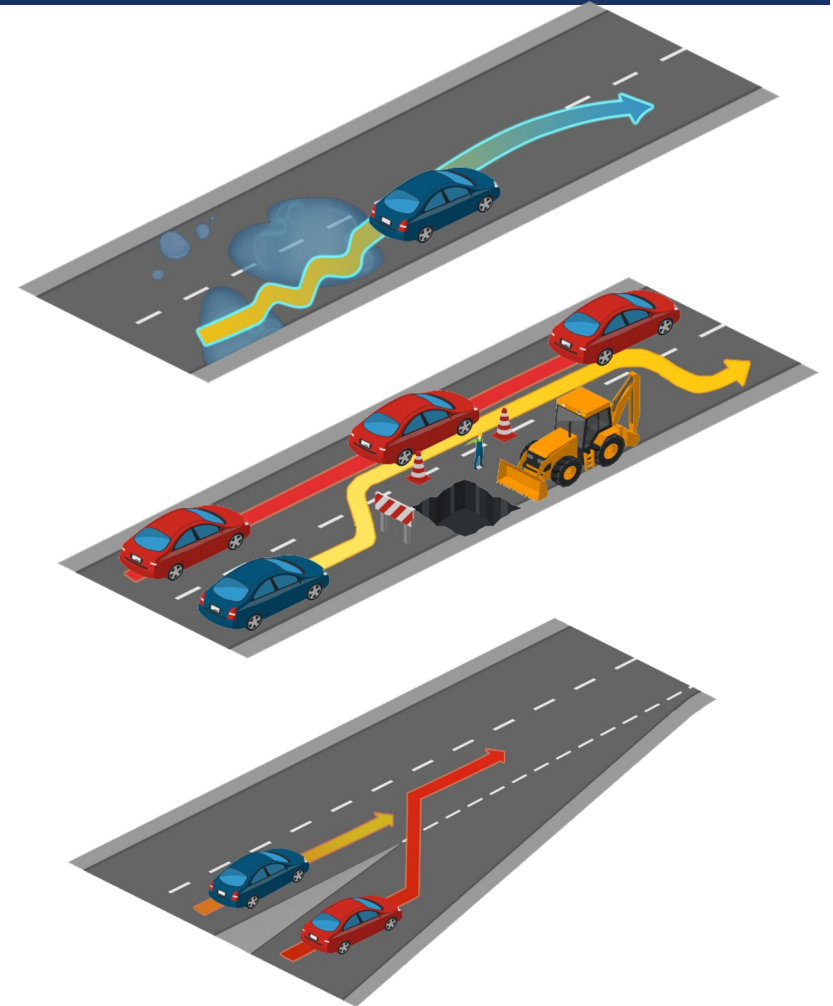
Coping with pedestrians, cyclists, and other elements.



PROJECTS SCENARIOS

The project consists of 10 Tracks from CARLA Leaderboard

- 5 Mandatory Tracks (routes_avddiem_exam.xml)
- 5 Optional Tracks (routes_avddiem_exam_extra.xml)



PROJECT EVALUATION

EVALUATION METRICS

DRIVING SCORE	Product between the route completion and the infraction penalty.
ROUTE COMPLETION	Percentage of the route distance completed by an agent.
INFRACTION PENALTY	Productory of the infractions committed . The base score is 1.0 and it is reduced at each infraction.

The average of each evaluation metric is computed when all routes have been completed.

They are named **GLOBAL METRICS**

GLOBAL DRIVING SCORE is the **MAIN METRIC**

THE INFRACTIONS

Infraction	Penalty	Description
Collisions with pedestrians	0.50	
Collisions with other vehicles	0.60	
Collisions with static elements	0.65	
Running a red light	0.70	
Running a stop sign	0.80	
Scenario Timeout	0.70	Vehicle stuck for 4 minutes. Raise Shutdown Event
Failure to maintain minimum speed	0.70	Fail to maintain the speed of other vehicles in traffic
Failure to yield to emergency vehicle	0.70	Fail to allow to pass an emergency vehicle
Off-road driving	No addition to Route Completion	

SHUTDOWN EVENTS

Shutdown Event	Description
Route Deviation	Deviation of almost 30 meters from assigned route
Agent Blocked	Agent does not take any actions for 180 seconds
Simulation Timeout	Client-Server communication cannot be established for 60 seconds
Route Timeout	Route takes too long to finish

A shutdown event **interrupts** the current route and the simulation passes to next one.

BEHAVIOR AGENT

BehaviorAgent implements an agent that navigates scenes to reach a given target destination, by computing the shortest possible path to it.

This agent can correctly follow traffic signs, speed limitations, traffic lights, while also taking into account nearby vehicles. Lane changing decisions can be taken by analyzing the surrounding environment such as tailgating avoidance.

Adding to these are possible behaviors, the agent can also keep safety distance from a car in front of it by tracking the instantaneous time to collision and keeping it in a certain range. Finally, different sets of behaviors are encoded in the agent, from cautious to a more aggressive ones.

RED LIGHTS AND STOPS

```
def _affected_by_traffic_light(self, lights_list=None, max_distance=None):
    """
    Method to check if there is a red light affecting the vehicle.

    :param lights_list (list of carla.TrafficLight): list containing TrafficLight objects.
        If None, all traffic lights in the scene are used
    :param max_distance (float): max distance for traffic lights to be considered relevant.
        If None, the base threshold value is used
    """
    if self._ignore_traffic_lights:
        return (False, None)

    if not lights_list:
        lights_list = self._world.get_actors().filter("*traffic_light*")

    if not max_distance:
        max_distance = self._base_tlight_threshold

    if self._last_traffic_light:
        if self._last_traffic_light.state != carla.TrafficLightState.Red:
            self._last_traffic_light = None
        else:
            return (True, self._last_traffic_light)

    ego_vehicle_location = self._vehicle.get_location()
    ego_vehicle_waypoint = self._map.get_waypoint(ego_vehicle_location)
```


RED LIGHTS AND STOPS (2)

```
for traffic_light in lights_list:
    if traffic_light.id in self._lights_map:
        trigger_wp = self._lights_map[traffic_light.id]
    else:
        trigger_location = get_trafficlight_trigger_location(traffic_light)
        trigger_wp = self._map.get_waypoint(trigger_location)
        self._lights_map[traffic_light.id] = trigger_wp

    if trigger_wp.transform.location.distance(ego_vehicle_location) > max_distance:
        continue

    if trigger_wp.road_id != ego_vehicle_waypoint.road_id:
        continue

    ve_dir = ego_vehicle_waypoint.transform.get_forward_vector()
    wp_dir = trigger_wp.transform.get_forward_vector()
    dot_ve_wp = ve_dir.x * wp_dir.x + ve_dir.y * wp_dir.y + ve_dir.z * wp_dir.z

    if dot_ve_wp < 0:
        continue

    if traffic_light.state != carla.TrafficLightState.Red:
        continue

    if is_within_distance(trigger_wp.transform,
                          self._vehicle.get_transform(),
                          max_distance, [0, 90]):
        self._last_traffic_light = traffic_light
        return (True, traffic_light)

return (False, None)
```

PEDESTRIAN AVOIDANCE

```
def pedestrian_avoid_manager(self, waypoint):
    """
    This module is in charge of warning in case of a collision
    with any pedestrian.

    :param location: current location of the agent
    :param waypoint: current waypoint of the agent
    :return vehicle_state: True if there is a walker nearby, False if not
    :return vehicle: nearby walker
    :return distance: distance to nearby walker
    """

    walker_list = self._world.get_actors().filter("*walker.pedestrian*")
    def dist(w): return w.get_location().distance(waypoint.transform.location)
    walker_list = [w for w in walker_list if dist(w) < 10]

    if self._direction == RoadOption.CHANGELANELEFT:
        walker_state, walker, distance = self._vehicle_obstacle_detected(walker_list, max(
            self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=90, lane_offset=-1)
    elif self._direction == RoadOption.CHANGELANERIGHT:
        walker_state, walker, distance = self._vehicle_obstacle_detected(walker_list, max(
            self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=90, lane_offset=1)
    else:
        walker_state, walker, distance = self._vehicle_obstacle_detected(walker_list, max(
            self._behavior.min_proximity_threshold, self._speed_limit / 3), up_angle_th=60)

    return walker_state, walker, distance
```

CAR AVOIDANCE

```
def collision_and_car_avoid_manager(self, waypoint):
    """
    This module is in charge of warning in case of a collision
    and managing possible tailgating chances.

    :param location: current location of the agent
    :param waypoint: current waypoint of the agent
    :return vehicle_state: True if there is a vehicle nearby, False if not
    :return vehicle: nearby vehicle
    :return distance: distance to nearby vehicle
    """

    vehicle_list = self._world.get_actors().filter("*vehicle*")
    def dist(v): return v.get_location().distance(waypoint.transform.location)
    vehicle_list = [v for v in vehicle_list if dist(v) < 45 and v.id != self._vehicle.id]

    if self._direction == RoadOption.CHANGELANELEFT:
        vehicle_state, vehicle, distance = self._vehicle_obstacle_detected(
            vehicle_list, max(
                self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=180, lane_offset=-1)
    elif self._direction == RoadOption.CHANGELANERIGHT:
        vehicle_state, vehicle, distance = self._vehicle_obstacle_detected(
            vehicle_list, max(
                self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=180, lane_offset=1)
    else:
        vehicle_state, vehicle, distance = self._vehicle_obstacle_detected(
            vehicle_list, max(
                self._behavior.min_proximity_threshold, self._speed_limit / 3), up_angle_th=30)

    # Check for tailgating
    if not vehicle_state and self._direction == RoadOption.LANEFOLLOW \
        and not waypoint.is_junction and self._speed > 10 \
        and self._behavior.tailgate_counter == 0:
        self._tailgating(waypoint, vehicle_list)

    return vehicle_state, vehicle, distance
```

OBJECT DETECTION

```
def _vehicle_obstacle_detected(self, vehicle_list=None, max_distance=None, up_angle_th=90, low_angle_th=0, lane_offset=0):
    """
    Method to check if there is a vehicle in front of the agent blocking its path.

    :param vehicle_list (list of carla.Vehicle): list containing vehicle objects.
        If None, all vehicle in the scene are used
    :param max_distance: max freespace to check for obstacles.
        If None, the base threshold value is used
    """
    if self._ignore_vehicles:
        return (False, None, -1)

    if not vehicle_list:
        vehicle_list = self._world.get_actors().filter("*vehicle*")

    if not max_distance:
        max_distance = self._base_vehicle_threshold

    ego_transform = self._vehicle.get_transform()
    ego_wpt = self._map.get_waypoint(self._vehicle.get_location())

    # Get the right offset
    if ego_wpt.lane_id < 0 and lane_offset != 0:
        lane_offset *= -1

    # Get the transform of the front of the ego
    ego_forward_vector = ego_transform.get_forward_vector()
    ego_extent = self._vehicle.bounding_box.extent.x
    ego_front_transform = ego_transform
    ego_front_transform.location += carla.Location(
        x=ego_extent * ego_forward_vector.x,
        y=ego_extent * ego_forward_vector.y,
    )
```

OBJECT DETECTION (2)

```
for target_vehicle in vehicle_list:
    target_transform = target_vehicle.get_transform()
    target_wpt = self._map.get_waypoint(target_transform.location, lane_type=carla.LaneType.Any)

    # Simplified version for outside junctions
    if not ego_wpt.is_junction or not target_wpt.is_junction:

        if target_wpt.road_id != ego_wpt.road_id or target_wpt.lane_id != ego_wpt.lane_id + lane_offset:
            next_wpt = self._local_planner.get_incoming_waypoint_and_direction(steps=3)[0]
            if not next_wpt:
                continue
            if target_wpt.road_id != next_wpt.road_id or target_wpt.lane_id != next_wpt.lane_id + lane_offset:
                continue

        target_forward_vector = target_transform.get_forward_vector()
        target_extent = target_vehicle.bounding_box.extent.x
        target_rear_transform = target_transform
        target_rear_transform.location -= carla.Location(
            x=target_extent * target_forward_vector.x,
            y=target_extent * target_forward_vector.y,
        )

        if is_within_distance(target_rear_transform, ego_front_transform, max_distance, [low_angle_th, up_angle_th]):
            return (True, target_vehicle, compute_distance(target_transform.location, ego_transform.location))
```

OBJECT DETECTION (3)

```
else:
    route_bb = []
    ego_location = ego_transform.location
    extent_y = self._vehicle.bounding_box.extent.y
    r_vec = ego_transform.get_right_vector()
    p1 = ego_location + carla.Location(extent_y * r_vec.x, extent_y * r_vec.y)
    p2 = ego_location + carla.Location(-extent_y * r_vec.x, -extent_y * r_vec.y)
    route_bb.append([p1.x, p1.y, p1.z])
    route_bb.append([p2.x, p2.y, p2.z])
    for wp, _ in self._local_planner.get_plan():
        if ego_location.distance(wp.transform.location) > max_distance:
            break
        r_vec = wp.transform.get_right_vector()
        p1 = wp.transform.location + carla.Location(extent_y * r_vec.x, extent_y * r_vec.y)
        p2 = wp.transform.location + carla.Location(-extent_y * r_vec.x, -extent_y * r_vec.y)
        route_bb.append([p1.x, p1.y, p1.z])
        route_bb.append([p2.x, p2.y, p2.z])

    if len(route_bb) < 3:
        # 2 points don't create a polygon, nothing to check
        return (False, None, -1)
    ego_polygon = Polygon(route_bb)

    # Compare the two polygons
    for target_vehicle in vehicle_list:
        target_extent = target_vehicle.bounding_box.extent.x
        if target_vehicle.id == self._vehicle.id:
            continue
        if ego_location.distance(target_vehicle.get_location()) > max_distance:
            continue

        target_bb = target_vehicle.bounding_box
        target_vertices = target_bb.get_world_vertices(target_vehicle.get_transform())
        target_list = [[v.x, v.y, v.z] for v in target_vertices]
        target_polygon = Polygon(target_list)

        if ego_polygon.intersects(target_polygon):
            return (True, target_vehicle, compute_distance(target_vehicle.get_location(), ego_location))
    return (False, None, -1)
```

CAR FOLLOWING

```
def car_following_manager(self, vehicle, distance, debug=False):
    """
    Module in charge of car-following behaviors when there's
    someone in front of us.

    :param vehicle: car to follow
    :param distance: distance from vehicle
    :param debug: boolean for debugging
    :return control: carla.VehicleControl
    """

    vehicle_speed = get_speed(vehicle)
    delta_v = max(1, (self._speed - vehicle_speed) / 3.6)
    ttc = distance / delta_v if delta_v != 0 else distance / np.nextafter(0., 1.)
    # Under safety time distance, slow down.
    if self._behavior.safety_time > ttc > 0.0:
        target_speed = min([
            positive(vehicle_speed - self._behavior.speed_decrease),
            self._behavior.max_speed,
            self._speed_limit - self._behavior.speed_lim_dist])
        self._local_planner.set_speed(target_speed)
        control = self._local_planner.run_step(debug=debug)
    # Actual safety distance area, try to follow the speed of the vehicle in front.
    elif 2 * self._behavior.safety_time > ttc >= self._behavior.safety_time:
        target_speed = min([
            max(self._min_speed, vehicle_speed),
            self._behavior.max_speed,
            self._speed_limit - self._behavior.speed_lim_dist])
        self._local_planner.set_speed(target_speed)
        control = self._local_planner.run_step(debug=debug)
    # Normal behavior.
    else:
        target_speed = min([
            self._behavior.max_speed,
            self._speed_limit - self._behavior.speed_lim_dist])
        self._local_planner.set_speed(target_speed)
        control = self._local_planner.run_step(debug=debug)

    return control
```

TAILGATING

[illegible]

OTHER BEHAVIORS

EMERGENCY STOP

```
control = carla.VehicleControl()
control.throttle = 0.0
control.brake = self._max_brake
control.hand_brake = False
return control
```

INTERSECTION BEHAVIOR

```
target_speed = min([
    self._behavior.max_speed,
    self._speed_limit - 5])
self._local_planner.set_speed(target_speed)
control = self._local_planner.run_step(debug=debug)
```

NORMAL BEHAVIOR

```
target_speed = min([
    self._behavior.max_speed,
    self._speed_limit - self._behavior.speed_lim_dist])
self._local_planner.set_speed(target_speed)
control = self._local_planner.run_step(debug=debug)
```

RESULT FILE

GLOBAL RECORD

Si può modificare dove salvare il file in run_test-.sh

INFRACTIONS

SCORES_MEAN

SCORE_COMPOSED : Global Driving Score

SCORE_ROUTE : Global Route Complention

SCORE_PENALTY : Global Infraction Penalty

SCORES_STD_DEV

SCORE_COMPOSED : StdDev Driving Score

SCORE_ROUTE : StdDev Route Complention

SCORE_PENALTY : StdDev Infraction Penalty

META

: Exceptions for each interrupted route

RESULT FILE

PROGRESS : Current Record Index to End Record Index

RECORDS : List of records

STATUS : Status of the record

INFRACTIONS : Detailed List of infraction

SCORES : Route Score

SCORE_COMPOSED (Driving Score)

SCORE_ROUTE (Route Complention)

SCORE_PENALTY (Infraction Penalty)

META : Route Lenght - Duration

BASELINE RESULTS

ROUTES_AVDDIEM_EXAM.XML

SCORE_COMPOSED : 7.89
SCORE_ROUTE : 36.34
SCORE_PENALTY : 0.52

**Execution Time : 101
min**

ROUTES_AVDDIEM_EXAM_RESULTS.JSON

ROUTES_AVDDIEM_EXAM_EXTRA.X ML

SCORE_COMPOSED : 8.16
SCORE_ROUTE : 31.10
SCORE_PENALTY : 0.20

**Execution Time : 105
min**

ROUTES_AVDDIEM_EXAM_EXTRA_RESULTS.JSON

PROJECT IN A NUTSHELL

GOAL	→	Run and evaluate your agent in the provided tracks
ROUTES_AVDDIEM_EXAM.XML	→	5 Required Tracks
ROUTES_AVDDIEM_EXAM_EXTRA.X	→	5 Optional Tracks
BASELINE ML	→	CARLA BEHAVIOR AGENT
EVALUATION METRIC	→	GLOBAL DRIVING SCORE
ROUTES_AVDDIEM_EXAM_RESULTS.JSON ROUTES_AVDDIEM_EXAM_EXTRA_RESULTS.JSON	→	BASELINE RESULTS

BASELINE

<https://drive.google.com/file/d/1-RnXxHVIVUCp1tgZZFk0v99URgvlUQ2Y/view?usp=sharing>