# AUTONOMOUS DRIVING

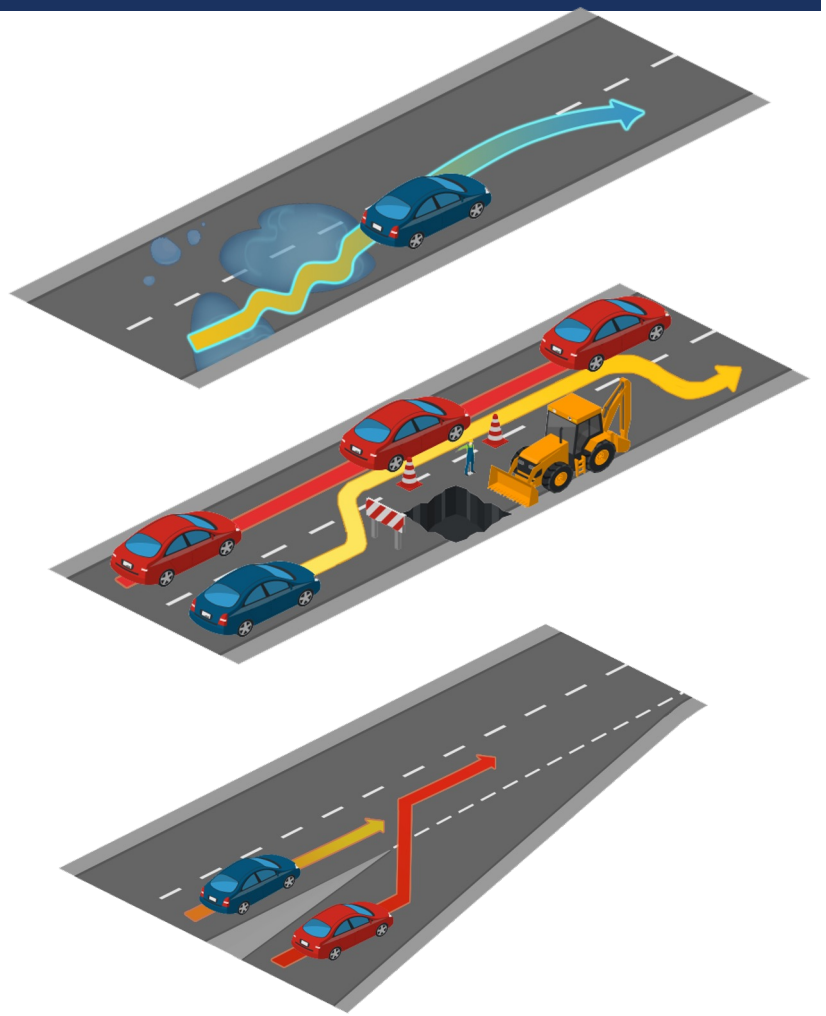
PROJECT

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.



# 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.

**BASELINE** https://drive.google.com/file/d/1-RnXxHVlVUCp1tgZZFk0v99URgvIuQ2Y/view?usp=sharing

# 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)
```

https://carla.readthedocs.io/en/latest/core_map/#waypoints

# 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)
```

get the waypoint associated with this traffic light from the map.

Check if the distance from the traffic light waypoint to the vehicle's location exceeds a maximum distance.

Check if the traffic light and the vehicle are not on the same road.

Calculate the direction vectors from the vehicle's waypoint and the traffic light's waypoint.

Compute the dot product of the two vectors to determine their alignment

Check if the traffic light is within a specified distance and angle from the vehicle.

If conditions are met, store the last seen traffic light and return a tuple indicating detection and the traffic light object.

self._lights_map = {}  # Dictionary mapping a traffic light to a wp corresponding to its trigger volume location

https://carla.readthedocs.io/en/latest/tuto_M_custom_add_tl/

# 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
```

The code checks the vehicle's intended direction, which is stored in self._direction

Method to check if there is a vehicle in front of the agent blocking its path.

Up_angle_th is used in'angle_interval', the angle between the location and reference transform will also be taken into account, being 0 a location in front and 180, one behind.

# 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
```

Method to check if there is a vehicle in front of the agent blocking its path.

# 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 contatining 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,
        )
```

Retrieves the waypoint associated with the vehicle's location on the map. A waypoint typically contains metadata about the part of the road where the vehicle is located, including lane information.

https://github.com/carla-simulator/carla/issues/1469

https://publications.pages.asam.net/standards/ASAM_OpenDRIVE/ASAM_OpenDRIVE_Specification/latest/specification/11_lanes/11_04_lane_offset.html

# 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)
    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

```
def _tailgating(self, waypoint, vehicle_list):
    """
    This method is in charge of tailgating behaviors.

        :param location: current location of the agent
        :param waypoint: current waypoint of the agent
        :param vehicle_list: list of all the nearby vehicles
    """

    left_turn = waypoint.left_lane_marking.lane_change
    right_turn = waypoint.right_lane_marking.lane_change

    left_wpt = waypoint.get_left_lane()
    right_wpt = waypoint.get_right_lane()

    behind_vehicle_state, behind_vehicle, _ = self._vehicle_obstacle_detected(vehicle_list, max(
        self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=180, low_angle_th=160)
    if behind_vehicle_state and self._speed < get_speed(behind_vehicle):
        if (right_turn == carla.LaneChange.Right or right_turn ==
                carla.LaneChange.Both) and waypoint.lane_id * right_wpt.lane_id > 0 and right_wpt.lane_type == carla.LaneType.Driving:
            new_vehicle_state, _, _ = self._vehicle_obstacle_detected(vehicle_list, max(
                self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=180, lane_offset=1)
            if not new_vehicle_state:
                print("Tailgating, moving to the right!")
                end_waypoint = self._local_planner.target_waypoint
                self._behavior.tailgate_counter = 200
                self.set_destination(end_waypoint.transform.location,
                                     right_wpt.transform.location)
        elif left_turn == carla.LaneChange.Left and waypoint.lane_id * left_wpt.lane_id > 0 and left_wpt.lane_type == carla.LaneType.Driving:
            new_vehicle_state, _, _ = self._vehicle_obstacle_detected(vehicle_list, max(
                self._behavior.min_proximity_threshold, self._speed_limit / 2), up_angle_th=180, lane_offset=-1)
            if not new_vehicle_state:
                print("Tailgating, moving to the left!")
                end_waypoint = self._local_planner.target_waypoint
                self._behavior.tailgate_counter = 200
                self.set_destination(end_waypoint.transform.location,
                                     left_wpt.transform.location)
```

# 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)
```

# BASELINE PARAMETERS

"longitudinal_control_dict" :{"K_P": 0.888, "K_I": 0.0768, "K_D": 0.05, "dt": 0.05},

- "lateral_control_dict" : {"K_V": 4, "K_S": 1, "dt": 0.05},

These values can be changed to improve the performance

# RESULT FILE

**GLOBAL RECORD**

- **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