carnd_t3_p1_path_planning_project

Carnd - term 3 - project 1 - path planning project

Overview

The goal of this project is to build a path planner model to drive vehicle around a 3-lane track and change lane as it sees opportunity to go ahead of other vehicles in front. The lane change action has to be performed safely without bumping into other vehicle as well as causing crash. The vehicle acceleration and turning has to be performed with minimal jerk to ensure comfort level for the passengers. The model processes a series of sensor data (x,y,s,d,yaw,speed,previous-path_x,previous_path_y,end_path_s,end_path_d,sensor_fusion,etc) to track the reference path and to predict the vehicle drive path, including lane changing. Spline library is used to optimize the waypoint fitting of the predicted reference path. Similar to other pervious projects in Term2, the path planning model is connected to a simulator via uWebSocketIO with waypoints of the reference path.

Project Repository

All resource are located in Udacity's project repository CarND-Path-Planning-Project (https://github.com/udacity/CarND-Path-Planning-Project)

Project Submission

All modified code including results are committed to my personal github page carnd_t3_p1_path_planning_project (https://github.com/chriskcheung/carnd_t3_p1_path_planning_project)

Key Files

main.cpp

establishes communication between simulator and path_planning model using uWebSocketIO, and reads in sensor data during a set time intervals. Data is passed into path planning class for initialization, processing sensor data, updating finite state machine (FSM) of path planner, and generating trajectory of the new predicted path.

path_planner.h and path_planner.cpp

contains 1 main class: pathPlanning.

pathPlanning class receives sensor data periodically. Base on the new sensor data, pathPlanning loops through all other vehicles data, for their locations and speed, then calculates the latest closest distance between other vehicles with respect to the subjected car, the cost functions of each lane with respect to their closest vehicle in both front and back.

There are 5 major functions involved: initPathPlanning(), processSensorData(), isChangeLaneSafe(), updateFSM(), generateTrajectory().

initPathPlanning() initializes all costs and predicted waypoints back to their default state or zero at every new data receives to start fresh.

processSensorData() loops through all other vehicles data from sensor_fusion to find the closest vehicle to the subjected car in each lane, considering both front and back. This maps out the landscape of the surround and ignores those that are outside of laneVisibleDist_ to avoid unnecessary tracking. Base on their s distance to the subjected car, and the velocity difference, cost_[] of changing lane into each lane is established for FSM to make decision.

isChangeLaneSafe() determines if it is safe to proceed from the existing lane to the targeting lane base on other vehicles position. The min_safe_dist_front_ and min_safe_dist_back_ ensure that there is enough room to the front and back vehicle on the left and the right before signaling safe to proceed.

updateFSM() tracks its current state of the subjected car. With the lowest cost_[*] and isChangeLaneSafe(), it determine which direction the car should turn, (target_lane_), vs staying in the current lane. The FSM also controls the speed of the car to either catch up or slow down to match the speed of vehicle in front to avoid collision, or the max legal speed. maxTravelSpeed_.

generateTrajectory() reuses up to 10 previous_path_x and previous_path_y waypoints to ensure smooth lane change transition. New way points are added from the 11th previous_path_* alone with new points from the spline fitting line to ensure a smooth fitting line for lane change.

Project Setup

I create an AWS instance to execute pathPlanning executible and run term3_sim simulator in local PC. Data is sent through AWS server to local PC client, which create latency. The latency is obvious during implementation when switching between wired and WiFi connections. I end up tuning my model with wired connect to keep things consistent.

Implementation Challenge

Cost function

Determine the right cost function is important to for the FSM to make decision. I start dividing the distance between subjected car and the closest vechicle in front of each lane (closest_front_s_diff_[*]) by the delta of velocity between two cars (ref_v_ - closest_front_v_[*]) to determine the time the two vehicle apart. The larger the time is the bigger the buffer and the less cost it should be. With exponential function on the result of the division, it doesn't reveal such relationships. Reciprocal of exponential results seems to provides that relationship, but the spread is too heavy on one side rather than evenly distributed. After a few try, I settle with 100-exp(x) where x is (closest_front_s_diff_[*])/(ref_v_ - closest_front_v_[*]). It provides the gradual increase of weight as x decrease which fits the bill of what I am looking for.

	exp(x)	exp(-x)	1-exp(-x)	100-exp(x)
1	2.718281828	0.367879441	0.632120559	97.28171817
2.5	12.18249396	0.082084999	0.917915001	87.81750604
2.826	16.87240086	0.059268388	0.940731612	83.12759914
3.547	34.71068165	0.028809575	0.971190425	65.28931835

I filter out any vehicles that are fall into the following conditions to simply the cost calculation:

1. for vehicle outside of the visible distance, cost will be 0 to represent an open lane for advancing

2. for vehicle that is 2 lanes away from current lane, cost set cost to 100 as it is not impossible to jump 2 lanes without crossing the adjacent lane

```
    // use the end of previous path info to determine the cost of changing l
ane
    switch(int(ref_lane_)){
        // left most lane
        case 0:
            // cost of staying in same lane base on the distance of vehicle
 in the front
            cost_[0] += closest_front_s_diff_[0] > laneVisibleDist_ ? 0 :
                        (ref_v_ - closest_front_v_[0])==0 ? 100 :
                        max(0.0, min(100.0 - exp(closest_front_s_diff_[0]/(r
ef_v_ - closest_front_v_[0])), 100.0));
            // cost of turn right base on any room on the right front
            cost_[1] += closest_front_s_diff_[1] > laneVisibleDist_ ? 0 :
                        (ref_v_ - closest_front_v_[1])==0 ? 100 :
                        max(0.0, min(100.0 - exp(closest_front_s_diff_[1]/(r
ef_v_ - closest_front_v_[1])), 100.0));
            // not supporting to jump 2 lanes, so max cost for right most la
ne
            cost_[2] += 100;
            break;
        ...
    }
}
```

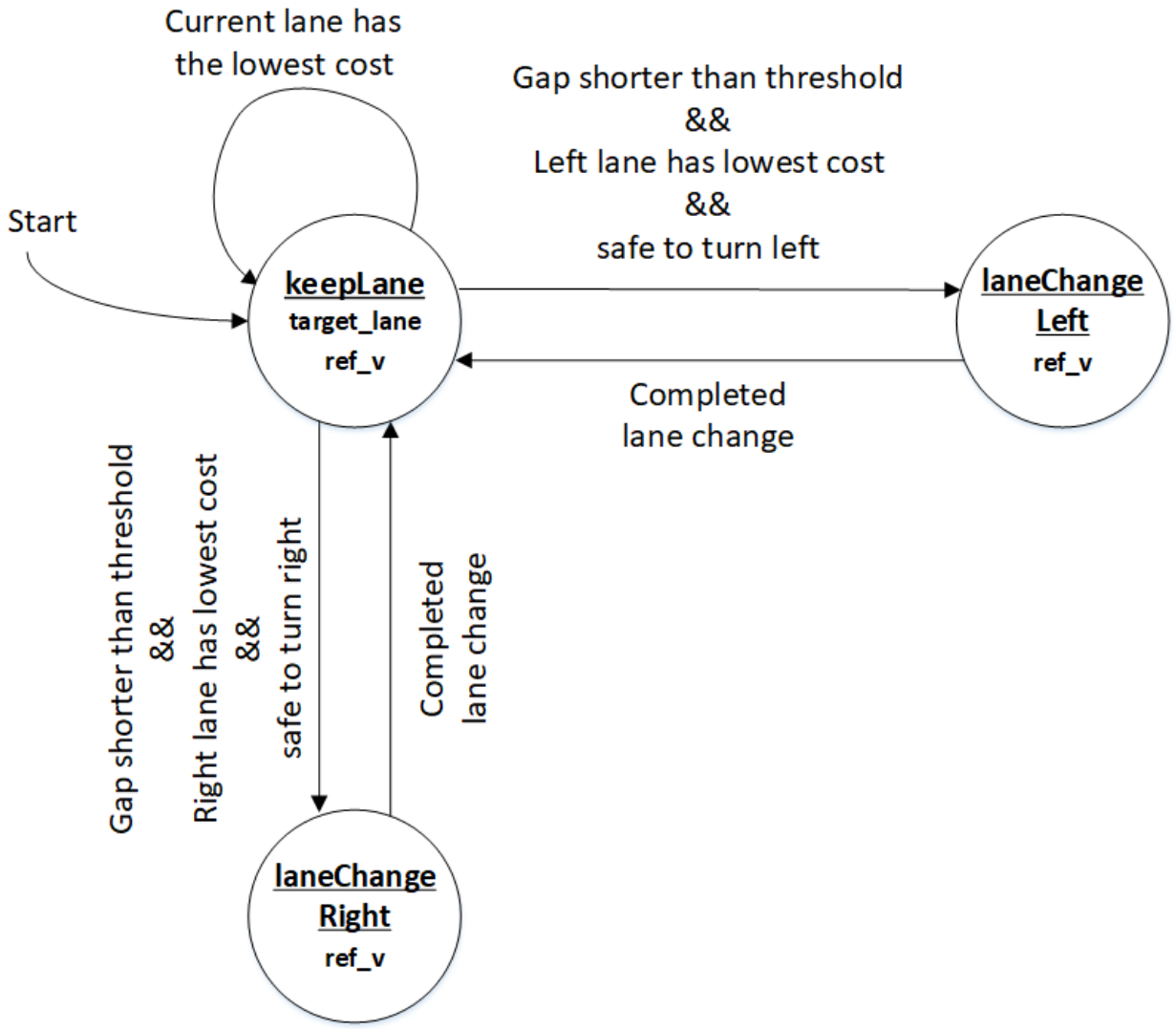
isChangeLaneSafe()

Checking whether it is safe to change lane depends on the subject vehicle position with respect the other vehicle in the front or in the back on current lane and the adjacent lane that is targeting for lane change. We have to look ahead of both vehicle in future (reference point from the previous 10th waypoint) to make sure they won't collide to each other when lane change actually happens. closest_front_s_diff_ subtracts previous_path_x[10] with future distance of other vehicle in adjacent lane at time of previous_path_x[10]. Same goes to vehicle from the back for each lane.

```
            // cap the previous point to be reused at the 10th point for smo
othing
            prv_size_  = min(int(previous_path_x.size()), 10);
            ...
            double prev_x1 = prv_x_[prv_size_-1];
            ...
            ref_x_ = prev_x1;
            ...

            double time_2_prev_path_end = dt_ * prv_size_;
            // vehicle velocity
            double vehicle_v  = sqrt(vehicle_vx*vehicle_vx + vehicle_vy*vehi
cle_vy);
            // vehicle s distance with respect to end of subjected car's pre
vious path trajectory
            double vehicle_s_at_prev_path_end = vehicle_s + vehicle_v*time_2
_prev_path_end;
            // s diff between subjected car and vehicle from sensor fusion
            double s_diff = fabs(ref_s_ - vehicle_s_at_prev_path_end);
            ...
            closest_front_s_diff_[vehicle_lane] = s_diff;
            ...
            if (closest_front_s_diff_[1] > min_safe_dist_front_ && closest_b
ack_s_diff_[1] > min_safe_dist_back_)
                return true;
            else
                return false;
```

FSM

This implementation is based on a 3 state finite state machine: keepLane, laneChangeLeft, and laneChangeRight states.



It starts at keepLane state. The state transition is based on 3 conditions: 1) if gap between the front vehicle is at or shorter than the minimum distance to consider lane change, 2) if cost of changing lane lower than cost of staying in current lane, 3) if it is safe to change lane to the targeting lane. keepLane state determines the target_lane_, and then pass it to the next state. laneChangeLeft and laneChangeRight both monitors the current lane from sensor data. Once the current lane matches with target_lane_, which signals that the lane change is completed successfully, it will transition back to keepLane state. Each state is responsible to maintain the subject vehicle speed to accelerate or decelerate its speed to match the front vehicle speed of the target_lane_ if presence, or catch up to legal speed limit. Speed should not exceed legal limit at any given time, thus speed will be decelerate when it gets too close to limit.

```
    case fsmStateType::keepLane:
            // consider change lane if gap between car in front is closer th
an minium lane change distance
            if(closest_front_s_diff_[ref_lane_] < min_lane_change_dist_){

                // reduce speed first to match the speed of car in front
                ref_v_ = max(ref_v_-0.6, closest_front_v_[ref_lane_]-0.6);

                // skip changing lane if it just finished changing lane with
in 10th of sec
                if(timer_ == 0){
                    //cost of changing lane
                    if(ref_lane_==0 && isChangeLaneSafe(0, 1) && (cost_[0] >
 cost_[1])){
                        f_ = fsmStateType::laneChangeRight;
                        target_lane_ = 1;
                        break;
                    }
                    else if(ref_lane_==2 && isChangeLaneSafe(2, -1) && (cost
_[2] > cost_[1])){
                        f_ = fsmStateType::laneChangeLeft;
                        target_lane_ = 1;
                        break;
                    }
                    else if(ref_lane_==1){
                        // change to left if cost of left lane is the lowest
 and if it is feasible
                        if(cost_[0] <= cost_[2] && cost_[0] < cost_[1] && is
ChangeLaneSafe(1, -1)){
                            // change lane left
                            f_ = fsmStateType::laneChangeLeft;
                            target_lane_ = 0;
                            break;
                        }
                        // change to right if cost of right lane is the lowe
st and if it is feasible
                        else if(cost_[2] <= cost_[0] && cost_[2] < cost_[1]
&& isChangeLaneSafe(1, 1)){
                            // change lane right
                            f_ = fsmStateType::laneChangeRight;
                            target_lane_ = 2;
                            cout << "SSS: change lane from 1 to RIGHT to " <
< target_lane_<< endl;
                            break;
                        }
                        // else stay in its lane if cost of its lane is the
 lowest
                    }
                }
            }
```

```
            else if(ref_v_ < maxTravelSpeed_*MILE_PER_HOUR_2_METER_PER_SEC)
                ref_v_ = min(ref_v_+0.8, maxTravelSpeed_*MILE_PER_HOUR_2_MET
ER_PER_SEC);
            else if (ref_v_ > maxTravelSpeed_*MILE_PER_HOUR_2_METER_PER_SEC)
                ref_v_ = max(ref_v_-0.6, maxTravelSpeed_*MILE_PER_HOUR_2_MET
ER_PER_SEC);

            if(timer_ < 0)
                timer_ = 0;
            else if(timer_ > 0)
                timer_ -= dt_*(nextPathSize_-prv_size_);
            target_lane_ = ref_lane_;
    break;

    case fsmStateType::laneChangeLeft:
            // lane change completed
            if(car_lane_ == target_lane_){
                f_ = fsmStateType::keepLane;
                // start timer to settle in to new lane before another lane
 change
                timer_ = 2;
            }

            // set targeting speed to either match the speed limit or the ve
hicle in front of target lane
            if(closest_front_v_[target_lane_] > 0 && closest_front_v_[target
_lane_] < target_v)
                target_v = closest_front_v_[target_lane_];

            // adjust speed
            if(ref_v_ < target_v)
                // purposely lower acceleration to +0.6 from +0.8 to reduce
 jerk when changing lane at the curve
                ref_v_ = min(ref_v_+0.6, target_v);
            else
                // purposely increase decceleration to -0.6 from -0.8 to red
uce jerk when changing lane at the curve
                ref_v_ = max(ref_v_-0.8, target_v);
    break;

    case fsmStateType::laneChangeRight:
            // complete lane change
            if(car_lane_ == target_lane_){
                f_ = fsmStateType::keepLane;
                // start timer to settle in to new lane before another lane
 change
                timer_ = 2;
            }
            // set targeting speed to either match the speed limit or the ve
hicle in front of target lane
```

```
            if(closest_front_v_[target_lane_] > 0 && closest_front_v_[target
_lane_] < target_v)
                target_v = closest_front_v_[target_lane_];
            if(ref_v_ < target_v)
                // purposely lower acceleration to +0.6 from +0.8 to reduce
 jerk when changing lane at the curve
                ref_v_ = min(ref_v_+0.6, target_v);
            else
                // purposely increase decceleration to -0.6 from -0.8 to red
uce jerk when changing lane at the curve
                ref_v_ = max(ref_v_-0.8, target_v);
    break;
```

State Machine Improvement

Notice that both laneChangeLeft and laneChangeRight states are identical. Since the target_lane_ is determined in keepLane and stays the same throughout laneChangeLeft and laneChangeRight, this FSM works just fine with 2 state implementation: keepLane and laneChange states. This will be for future improvement.

Jerk prevention

During lane change at straight lanes, jerk will not be an issue. However, when changing lane at the curve, with the extra turning on top of the curvature of the lane, it may apply additional force to create jerk. Simple tricks can be done to prevent this. It is by purpose to lower acceleration from 0.8 to 0.6 and increase deceleration from 0.6 to 0.8 to reduce speed as well as jerk create during lane change.

```
            if(ref_v_ < target_v)
                // purposely lower acceleration to +0.6 from +0.8 to reduce
 jerk when changing lane at the curve
                ref_v_ = min(ref_v_+0.6, target_v);
            else
                // purposely increase decceleration to -0.6 from -0.8 to red
uce jerk when changing lane at the curve
                ref_v_ = max(ref_v_-0.8, target_v);
```

Trajectory

Trajectory generation consists of two parts: reuse of preivous waypoints as the base points, and newly generated points from the last of the previous.

The reuse of previous waypoints is for the purpose of continuation from previous prediction to avoid a suddent change in direction and avoid jerk.

```
    // Since the spline fitting line starts from the last 2 points from the
 previous path points
    // as a continuation and the rest of the points are newly plotted/picked
 to predict the new path
    // for the next 30 meters. With the spline fitting line ready, break the
 spline evenly to N points
    // with the distance and time interval base on the the speed that we wan
t to travel
    // start with all of previous path points from last time
    for(int i=0; i<prv_size_; i++) {
        next_x.push_back(previous_path_x[i]);
        next_y.push_back(previous_path_y[i]);
    }
```

The new waypoint generation is based on the last two points from the reuse waypoints, plus three more points that are 30 meters apart to create a nice even sketch of the line. These five points are fitted into Spline equation to fit a line with nice and smooth curvature for our prediction.

Note

Previously, I use 30 meters as suggested by Arron Brown in his walkthrough video. I find that not enough to create a smooth fitting line that would avoid jerk. I could have blamed it to my vehicle speed being too fast. However, increasing the distance fo these three waypoints from 30 meters to 50 meters apart makes the spline fitting line even my smooth and the subject vehicle changes lane as smooth as lane splitting. Therefore, I leave it at 50 meters as my final submission version.

```
    // in Frenet add evenly 30m spaced points ahead of the starting referenc
e so the points are not only
    // cover the even distance points. original 30m apart, curently increase
 to 50m apart to further
    // reduce jerk during lane change by sketching out the spline fit line
    vector<double> next_wp0 = getXY(car_s_+50,  (2+laneWidth_*target_lane_),
 map_s_, map_x_, map_y_);
    vector<double> next_wp1 = getXY(car_s_+100, (2+laneWidth_*target_lane_),
 map_s_, map_x_, map_y_);
    vector<double> next_wp2 = getXY(car_s_+150, (2+laneWidth_*target_lane_),
 map_s_, map_x_, map_y_);
    ...
    ptsx.push_back(prv_x2);
    ptsy.push_back(prv_y2);
    ...
    // push the ref_x_ and ref_y_ as 2nd point, see processSensorData() for
 their origin
    ptsx.push_back(ref_x_);
    ptsy.push_back(ref_y_);

    ptsx.push_back(next_wp0[0]);
    ptsx.push_back(next_wp1[0]);
    ptsx.push_back(next_wp2[0]);

    ptsy.push_back(next_wp0[1]);
    ptsy.push_back(next_wp1[1]);
    ptsy.push_back(next_wp2[1]);
```

These coordinates are based in global coordinates. It is easier to convert them to car's coordinates (or Frenet coordinates) before fitting happens.

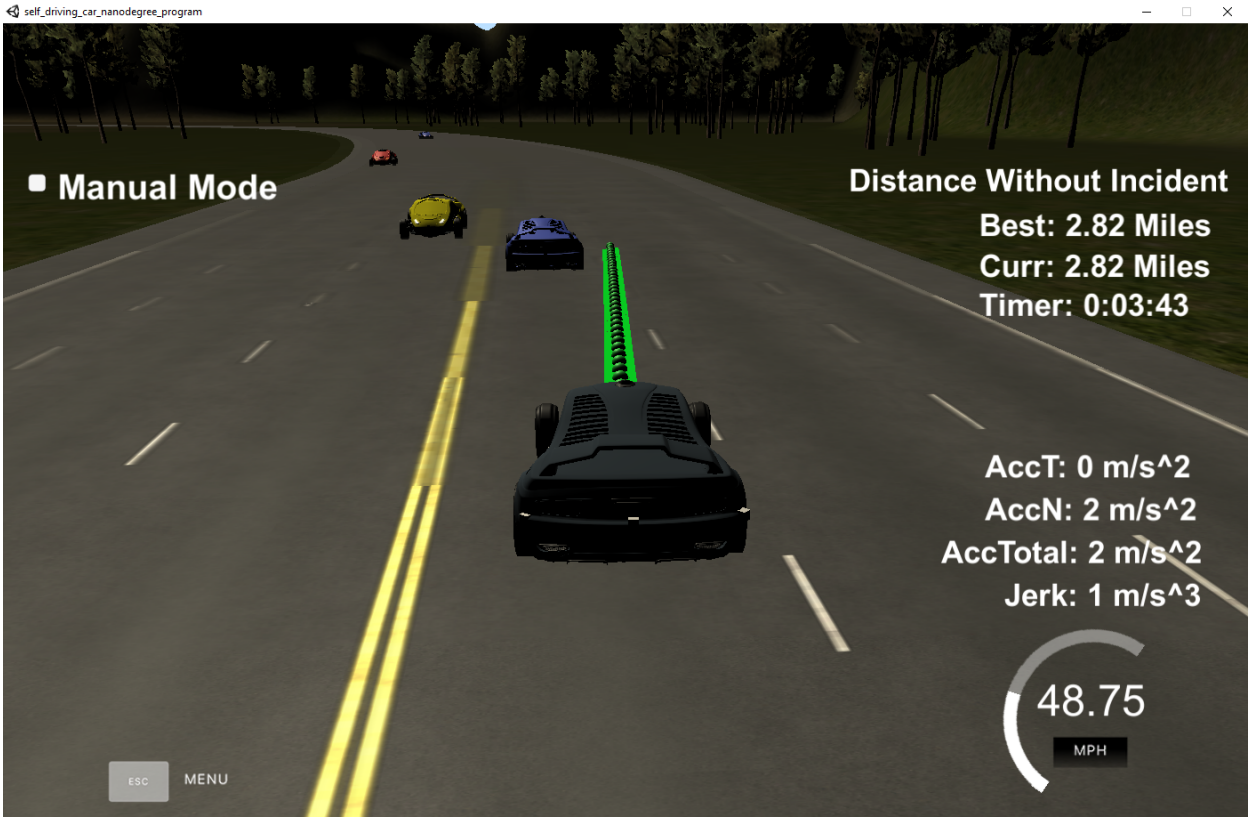
```
    for(int i=0; i<ptsx.size(); i++){
        //shift car reference angle to 0 degrees to align with local coordin
ates or in car's prespective
        double shift_x = ptsx[i]-ref_x_;
        double shift_y = ptsy[i]-ref_y_;

        ptsx[i] = (shift_x*cos(0-ref_yaw_)) - (shift_y*sin(0-ref_yaw_));
        ptsy[i] = (shift_x*sin(0-ref_yaw_)) + (shift_y*cos(0-ref_yaw_));
    }

    // create a spline for fitting
    tk::spline s;

    // set (x,y) points to spline
    s.set_points(ptsx, ptsy);
```

Once the spline fitting line is done, spread the points evenly over a distance of 30 meters, using the target speed, ref_v_, that we obtain from FSM. It is important to use ref_v_ as it how fast the subject vehicle should go to avoid colliding into the vehicle in front. Also, each waypoint has to be converted back to global coordinates before feeding back to simulator to drive the subject vehicle.

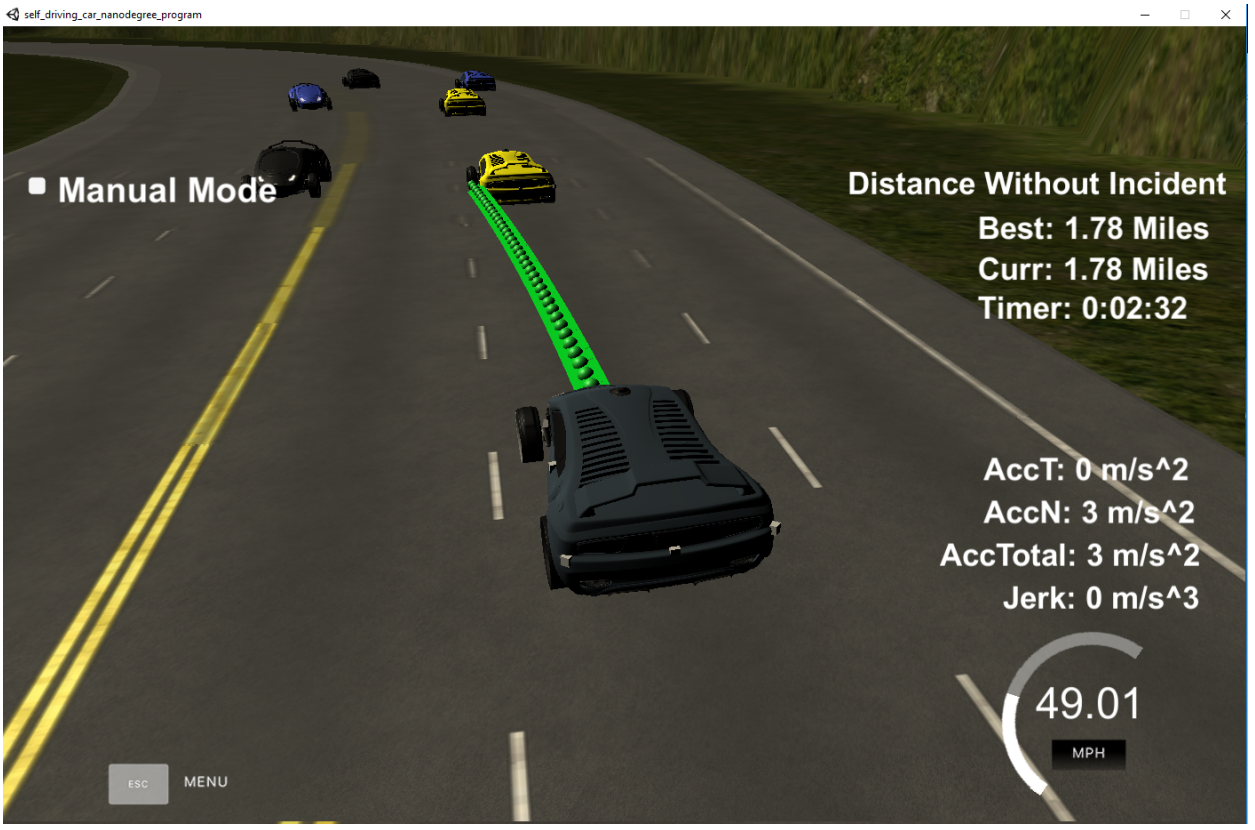
```
    // fill up the rest of out path planner after filling it with previous p
oints, here we will always output 50 points
    double N = (target_dist/(0.02*ref_v_));  // ref_v base on meter/sec

    for(int i=1; i<=nextPathSize_-prv_size_; i++){
        double x_point = x_add_on+(target_x/N);
        double y_point = s(x_point);

        x_add_on = x_point;

        double x_ref = x_point;
        double y_ref = y_point;

        // rotate back to normal after rotating it eariler (back to global c
oordinates)
        x_point = (x_ref*cos(ref_yaw_)) - (y_ref*sin(ref_yaw_));
        y_point = (x_ref*sin(ref_yaw_)) + (y_ref*cos(ref_yaw_));

        x_point += ref_x_;
        y_point += ref_y_;

        next_x.push_back(x_point);
        next_y.push_back(y_point);
    }
```

Results

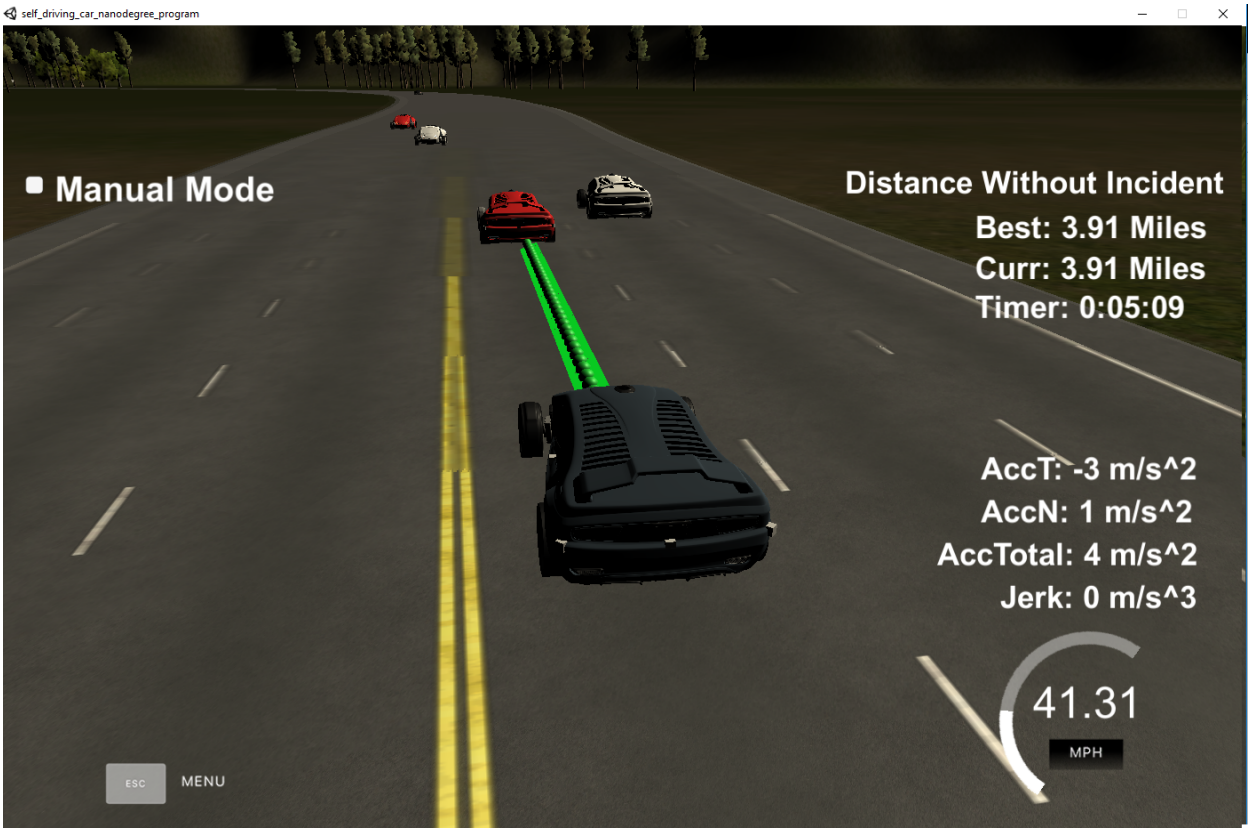
Unlike all previous project, this project is quite challenging as it is not given a lot of hints or guidance to begin with. However, all materials are covered by lecture one way or the other. It is up to us the student to put them together. There are many ways to approach the problem set and I found this very interesting when working out all the piece to finalize my solutions with 3 state FSM or even possibly to improve to a 2 state FSM. The result is quite satisfactory as one of my best lap reaches 46 miles without instance. The challenging part is to tuning the safety distance, speeds, and even the newly generated waypoints to avoid jerk when changing lanes. Below, I capture a few of the screenshot to illustrate the lane change moments and the best lap result.



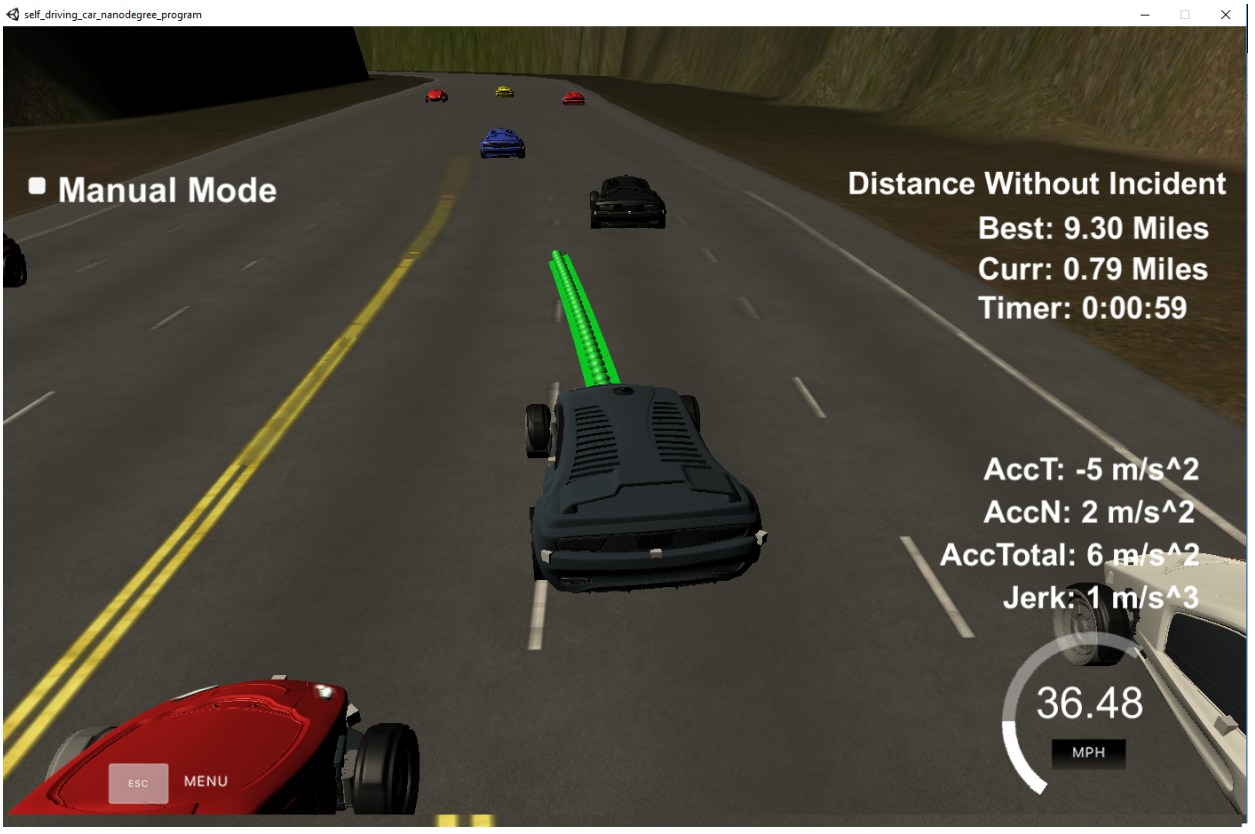
Making right turn



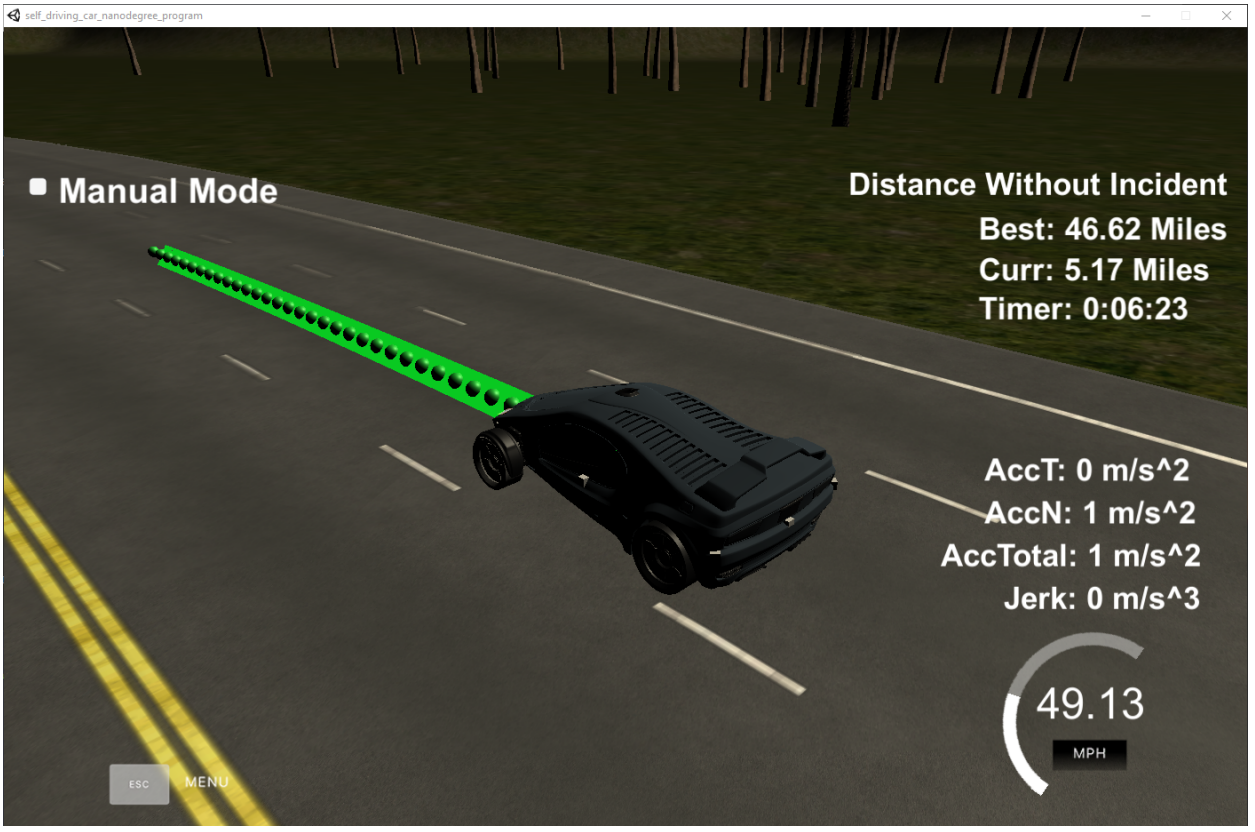
Marking left turn at the curve



Two cars in parallel in front, car turns to middle lane, then the right most lane



Black vehicle cut in front, car makes left turn right away



Best lap of 46 miles with incidence