Path-Planning

Write up

1st submit: September 12th, Kenta Kumazaki

1. Goals

In this project my goal is to safely navigate around a virtual highway with other traffic that is driving +-10 MPH of the 50 MPH speed limit. I'm provided the car's localization and sensor fusion data, there is also a sparse map list of waypoints around the highway. The car should try to go as close as possible to the 50 MPH speed limit, which means passing slower traffic when possible, note that other cars will try to change lanes too. The car should avoid hitting other cars at all cost as well as driving inside of the marked road lanes at all times, unless going from one lane to another. The car should be able to make one complete loop around the 6946m highway. Since the car is trying to go 50 MPH, it should take a little over 5 minutes to complete 1 loop. Also the car should not experience total acceleration over 10 m/s^2 and jerk that is greater than 10 m/s^3.

2. Steps

The steps of this project are the following:

(1) Drive the course with Lane Keep mode. (main.cpp)

I added some features as below:

- Change the ego vehicle speed gradually to avoid overshoot of acceleration / jerk limitations.
- If the preceding vehicle's position is close to the ego vehicle, speed down to avoid collision.
- If the vehicle on the next lane tries to Lane Change to current lane, speed down to avoid collision.
- If the behind vehicle is too close to the ego vehicle, speed up to avoid collision.

(2) Calculate the cost functions on each lane and find the optimal one that has the lowest cost. (lanechange.cpp)

I set the following cost functions:

- Static cost on each lane. Left lane has a smaller cost because basically other vehicles tend to drive faster in the left lane.
- Traffic density cost on each lane. The less other vehicles in front of the ego vehicle, the less the cost becomes.
- Average speed cost on each lane. The higher other vehicles run in front of the ego vehicle, the less the cost becomes.
- Distance cost between the preceding vehicle. If it's too close to the ego vehicle, the cost becomes high to encourage Lane Change to other lanes.

(3) Execute Lane Change if the current lane isn't the optimal one. (main.cpp)

I added some features as below:

- Set 1 second counter to change the lane to avoid hunting.
- Execute Lane Change only when it's considered safe.

3. Submission

(1) GitHub

https://github.com/kkumazaki/Self-Drivig-Car_Project7_Path-Planning.git

(2) Directory

I cloned the basic repository from Udacity https://github.com/udacity/CarND-Path-Planning-Project and added/modified the following files.

- **Writeup_of_Lesson11.pdf**: This file
- **src**
  - **main.cpp**: Main script to run functions and communicate with Simulator.
  - **lanechange.cpp**: Script used to create the function to judge Lane Change according to the cost functions.
  - **lanechange.h**: Header file of "lanechange.cpp".
  - **spline.h**: Script to make a smooth path. Download file from the recommended homepage as below: https://kluge.in-chemnitz.de/opensource/spline/

4. Reflection

(1) Drive the course with Lane Keep mode. (main.cpp)

In “main.cpp”, I added the following codes to drive the ego vehicle smoothly in the Simulator.

First of all, I check the sensor fusion data to check the following things to avoid collision. **(A)**

- If the preceding vehicle’s location in the same lane is close to the ego vehicle, set the flag to decelerate. I set 2 ranges of the distance to control finely. **(B)**
- If the behind vehicle is too close to the ego vehicle, set the flag to accelerate. **(C)**
- If the vehicle on the next lane tries to Lane Change to my lane, set the flag to decelerate. **(D)**
- If the preceding vehicle suddenly & strongly decelerate, set the flag to decelerate. **(E)**

```cpp
 * TODO: define a path made up of (x,y) points that the car will visit
 *   sequentially every .02 seconds
 */

/* [Step2: Smoothing the Path]*/

// Previous path size
int prev_size = previous_path_x.size();

/* [Step3: Sensor Fusion]*/
if (prev_size > 0){
  car_s = end_path_s;
}

bool too_close1 = false; // Flag for deceleration at regular distance (preceding vehicle)
bool too_close2 = false; // Flag for strong deceleration at short distance (preceding vehicle)
bool too_close3 = false; // Flag for strong acceleration at short distance (behind vehicle)
bool too_close4 = false; // Flag for strong deceleration for possible other car's Lane Change
bool too_close5 = false; // Flag for strong deceleration for sudden deceleration (preceding vehicle)

// Find ref_v to use
for (int i = 0; i < sensor_fusion.size(); i++){
  // Other cars
  float d = sensor_fusion[i][6];
  double vx = sensor_fusion[i][3];
  double vy = sensor_fusion[i][4];
  double check_speed = sqrt(vx*vx + vy*vy);
  double check_car_s = sensor_fusion[i][5];

  // Check the car in the same lane
  if (d < (2 + 4*lane + 2) && d > (2 + 4*lane -2)){
    // Predict the future s position of the car
    check_car_s += ((double)prev_size * sample_time * check_speed);

    // If the car in front of us is close to our car, we should decelerate
    //if((check_car_s > car_s) && ((check_car_s - car_s) < 30)){
    if((check_car_s > car_s) && ((check_car_s - car_s) < 40)){ // too avoid collision when preceding vehicle suddenly stops or other vehicle cutf in
      /*[Step4] Improve smoothness of deceleration and acceleration*/
      //ref_vel = 29.5;
      too_close1 = true;
    }
    //if((check_car_s > car_s) && ((check_car_s - car_s) < 15)){
    if((check_car_s > car_s) && ((check_car_s - car_s) < 20)){ // too avoid collision when preceding vehicle suddenly stops or other vehicle cutf in
      too_close2 = true;
    }
    if((check_car_s < car_s) && ((check_car_s - car_s) > -30)){
      too_close3 = true;
    }
  }
  // If the car in the left lane tries to Lane Change to my lane, need deceleration
  if (d > (2 + 4*lane - 6) && d < (2 + 4*lane - 2)){
    std::cout << "Left vehicle's d: " << d << std::endl;
    if((check_car_s > car_s) && ((check_car_s - car_s) < 40) && d > (1.5 + 4*lane - 2)){
      too_close4 = true;
    }
  }

  // If the car in the right lane tries to Lane Change to my lane, need deceleration
  if (d > (2 + 4*lane + 2) && d < (2 + 4*lane + 6)){
    std::cout << "Right vehicle's d: " << d << std::endl;
    if((check_car_s > car_s) && ((check_car_s - car_s) < 40) && d < (2.5 + 4*lane + 2)){
      too_close4 = true;
    }
  }
  if((check_car_s > car_s) && ((check_car_s - car_s) < 100) && (check_speed - ref_vel/mph2ms) < -15){
    too_close5 = true;
  }

}
```

According to the flags above, I calculate the ego vehicle's speed as below.

I change the speed gradually (at most: 0.4 mph / 0.02sec) to avoid exceeding acceleration and jerk limitation.

```
if (too_close1){
  ref_vel -= 0.2; // regular deceleration
}
else if(too_close2){
  ref_vel -= 0.4; // strong deceleration
}
else if(too_close4){
  ref_vel -= 0.3; // strong deceleration
}

else if(ref_vel < 49.5){
  ref_vel += 0.3;
  if (too_close3){
    ref_vel += 0.1; // strong acceleration
  }
}
else if(too_close5){
  ref_vel -= 0.4; // strong deceleration
}
```

(2)Calculate the cost functions on each lane and find the optimal one that has the lowest cost. (lanechange.cpp)

I created a new file “lanechange.cpp” and “lanechange.h” in this project.

Output of the function “lane_change()” is the optimal lane that has the lowest cost.

I set the minimum velocity for Lane Change **(A)** because it may not be safe to change lanes at low speed when other vehicles drive fast.

I made 4 kinds of cost functions as below, and I set weight on each cost function.

Each cost function has different unit, so it’s difficult to decide the optimal weights.

I tried the Simulation a couple of times with different weights and finally decided them. **(B)**

- Static cost on each lane. Left lane has a smaller cost because basically other vehicles tend to drive faster in the left lane.
- Traffic density cost on each lane. The less other vehicles in front of the ego vehicle, the less the cost becomes.
- Average speed cost on each lane. The higher other vehicles run in front of the ego vehicle, the less the cost becomes.
- Distance cost between the preceding vehicle. If it’s too close to the ego vehicle, the cost becomes high to encourage Lane Change to other lanes.

About the 1st cost function, I set the static cost on each lane.

The left lane has the lowest cost (0.0), and the right lane has the highest cost (2.0). **(C)**

```
int lane_change(vector<vector<double>> &sensor_fusion, double car_s, double car_d, double car_speed, double ref_ve
  // Minimum velocity for lane change
  double lc_vel = 20; //(mph)

  // Transfer (mph) to (m/s)
  double mph2ms = 3600.0/1609.0;

  vector<int> cnt = {0, 0, 0};
  vector<double> average_vel = {0.0, 0.0, 0.0};
  vector<double> lane_cost = {0.0, 1.0, 2.0};

  vector<double> traffic_flow = {49.5, 45.0, 40.0};


  double weight_density = 5.0;
  double weight_velofity = 10.0;
  double weight_deceleration = 10.0;
  double weight_lanes = 3.0;

  double total_cost = 0.0;
```

To calculate the 2nd and 3rd cost, I get the other vehicle numbers and average speeds by using sensor fusion data. **(A)** I calculate the number and speed in each lane at the same time. **(B)**

```
for (int i = 0; i < sensor_fusion.size(); i++){
  float d = sensor_fusion[i][6];
  double vx = sensor_fusion[i][3];
  double vy = sensor_fusion[i][4];
  double check_speed = sqrt(vx*vx + vy*vy);
  double check_car_s = sensor_fusion[i][5];

  if ((check_car_s > car_s) && (check_car_s - car_s < 90.0 ) && (d < 4.0)){
    cnt[0] += 1;
    average_vel[0] += check_speed;
  } else if ((check_car_s > car_s) && (check_car_s - car_s < 90.0 ) && (d < 8.0)){
    cnt[1] += 1;
    average_vel[1] += check_speed;
  } else if ((check_car_s > car_s) && (check_car_s - car_s < 90.0 ) && (d < 12.0)){
    cnt[2] += 1;
    average_vel[2] += check_speed;
  }

}

// calculate average traffic velocity at i=0: left, i=1: center, i=2: right
for (int i = 0; i < 3; i++){
  if (cnt[i]>0){
    average_vel[i] = average_vel[i]/cnt[i];
  } else{
    average_vel[i] = traffic_flow[i] / mph2ms;
  }
}
```

I multiply the weight on the each cost function for each lane (left, center, right). **(C)**

For the 4th cost function, I only calculate it on the current lane. **(D)**

And then, I get the optimal lane that has the minimum cost. **(E)**

```
double total_cnt = cnt[0] + cnt[1] + cnt[2];

// [Cost Function]
for (int i = 0; i < 3; i++){
  // [Cost Function 1. Set the weighted cost on each lane]
  lane_cost[i] = lane_cost[i] * weight_lanes;

  // [Cost Function 2. Set the weighted cost on traffic density]
  if (total_cnt > 0.1){
    lane_cost[i] += weight_density * cnt[i]/total_cnt;
  }

  // [Cost Function 3. Set the weighted cost on average velocity]
  lane_cost[i] += weight_velofity * (49.5 - average_vel[i]*mph2ms);
}

// [Cost Function 4. Set the weighted cost if the preceding vehicle is too slow]
if(too_close1 == true){
  lane_cost[lane] += weight_deceleration;
}

std::cout << "Cost[0]: " << lane_cost[0] << " Cost[1]: " << lane_cost[1] << " Cost[2]: " << lane_cost[2] << std::endl;

double min_cost = 10000.;
int index = lane; // default: no lane change

// [Minimum Cost Function]
for (int i = 0; i < 3; i++){
  if (lane_cost[i] < min_cost){
    min_cost = lane_cost[i];
    index = i;
  }
}
```

If the vehicle speed is high enough **(A)** and there's no other vehicles near from the ego vehicle on the next lane, then finally judge to change the lane **(B)**.

As described later, I check the feasibility of Lane Change right before executing it.

So, when I judge the optimal lane here, I only use a simple condition (distance > 50m).

```
// Plan lane change if there's no other vehicles around the lc area
// Vehicle State: Lane Keep --> Prepare for Lane Change
for (int i = 0; i < sensor_fusion.size(); i++){
  float d = sensor_fusion[i][6];
  double vx = sensor_fusion[i][3];
  double vy = sensor_fusion[i][4];
  double check_speed = sqrt(vx*vx + vy*vy);
  double check_car_s = sensor_fusion[i][5];

  // Left lane chagne
  if (index - lane < 0 && ref_vel > lc_vel){
    //Check the car in the left lane
    if (d > (2 + 4*lane - 6) && d < (2 + 4*lane - 2)){
      //Predict the future s position of the car
      check_car_s += ((double)prev_size * sample_time * check_speed);

      // If the car in behind us is far from our car, we can chane lane
      if ((abs(check_car_s - car_s) > 50)){
        lane -= 1;
      }
    }
  }

  // Right lane chagne
  if (index - lane > 0 && ref_vel > lc_vel){
    //Check the car in the left lane
    if (d > (2 + 4*lane + 2) && d < (2 + 4*lane + 6)){
      //Predict the future s position of the car
      check_car_s += ((double)prev_size * sample_time * check_speed);

      // If the car in behind us is far from our car, we can chane lane
      if ((abs(check_car_s - car_s) > 50)){
        lane += 1;
      }
    }
  }
}

return lane;
```

(3) Execute Lane Change if the current lane isn’t the optimal one. (main.cpp)

Now I come back to “main.cpp” again.

Here I call “lane_change()” function, which was described above, and update the variable “lane”. **(A)**

Then I wait for “stable_count” (1sec * 50 counts/sec) to change the lane to avoid hunting. **(B)**

```
/*[Step5] Lane Change*/
// Save the current lane information
int current_lane = lane;

// Vehicle State
int vehicle_state = 0; // 0: Lane Keep, 1: Prepare for Lane Change, 2: Lane Change

lane = lane_change(sensor_fusion, car_s, car_d, car_speed, ref_vel, lane, prev_size, sample_time, too_close1);

std::cout << "target lane: " << lane << std::endl;

// Judge Lane change one by one after keeping stable time
// Vehicle State: Prepare for Lane Change
if (lane != current_lane){
  vehicle_state = 1; // Prepare for Lane Change
  lc_count += 1;
  if (lc_count < stable_count){
    lane = current_lane;
    //std::cout << "Waiting for stable count." << std::endl;
  } else{
    //std::cout << "After stable count." << std::endl;
    if(lane < current_lane){
      lane = current_lane - 1; // Reject direct lane change from Right to Left
    } else {
      lane = current_lane + 1; // Reject direct lane change from Left to Right
    }
    lc_count = 0;
  }
}
else{
  std::cout << "Terget lane is equal to Current lane." << std::endl;
  lc_count = 0;
}

std::cout << "lc_count: " << lc_count << std::endl;
```

Finally, I use sensor fusion data right before executing Lane Change. **(A)**

The checking area is the target lane. **(B)**

If there're other vehicles near from the ego vehicle (Distance < 30m) or the relative speed is high (Time To Collision < 3 sec), it may not be safe to execute Lane Change so reject it (lane = current_lane).

Otherwise, execute Lane Change as desired. **(C)**

```cpp
// Execute Lane change if it's safe to avoid collision
// Vehicle State: Prepare for Lane Change --> Lane Change
for (int i = 0; i < sensor_fusion.size(); i++){
  // Other cars
  float d = sensor_fusion[i][6];
  double vx = sensor_fusion[i][3];
  double vy = sensor_fusion[i][4];
  double check_speed = sqrt(vx*vx + vy*vy);
  double check_car_s = sensor_fusion[i][5];

  // Left lane change
  if (lane < current_lane){

    if (d > (2 + 4*current_lane - 6) && d < (2 + 4*current_lane - 2)){
      // Predict the future s position of the car
      check_car_s += ((double)prev_size * sample_time * check_speed);

      //Debug
      std::cout << "Other vehicle speed: " << check_speed * mph2ms << std::endl;
      std::cout << "Other vehicle speed difference: " << (ref_vel/mph2ms - check_speed) << std::endl;
      std::cout << "Other vehicle position difference: " << (check_car_s - car_s) << std::endl;
      std::cout << "TTC: " << abs((check_car_s - car_s)/(ref_vel/mph2ms - check_speed)) << std::endl;

      // If the car is close to our car or speed difference is large, reject lane change
      //if((abs(check_car_s - car_s) < 30) && (abs(check_speed * mph2ms - ref_vel) > 15)){
      if(abs(check_car_s - car_s) < 30.0 || (abs((check_car_s - car_s)/(ref_vel/mph2ms - check_speed)) < 3.0)){
        lane = current_lane;
        std::cout << "Reject Lane Change to avoid collision." << std::endl;
      } else {
        vehicle_state = 2; // Lane Change
      }
    }
  }
  // Right lane change
  else if (lane > current_lane){

    if (d > (2 + 4*current_lane + 2) && d < (2 + 4*current_lane + 6)){
      // Predict the future s position of the car
      check_car_s += ((double)prev_size * sample_time * check_speed);

      //Debug
      std::cout << "Other vehicle speed: " << check_speed * mph2ms << std::endl;
      std::cout << "Other vehicle speed difference: " << (ref_vel/mph2ms - check_speed) << std::endl;
      std::cout << "Other vehicle position difference: " << (check_car_s - car_s) << std::endl;
      std::cout << "TTC: " << (check_car_s - car_s)/(ref_vel/mph2ms - check_speed) << std::endl;

      // If the car is close to our car or speed difference is large, reject lane change
      if(abs(check_car_s - car_s) < 30.0 || (abs((check_car_s - car_s)/(ref_vel/mph2ms - check_speed)) < 3.0)){
        lane = current_lane;
        std::cout << "Reject Lane Change to avoid collision." << std::endl;
      } else {
        vehicle_state = 2; // Lane Change
      }
    }
  }
}

std::cout << "Final Lane: " << lane << std::endl;
```

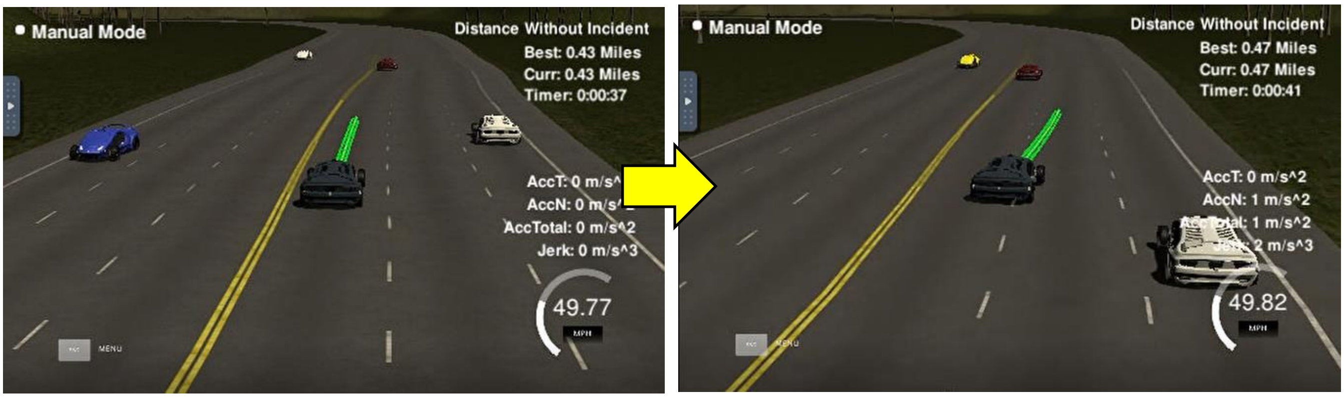
5. Result

I ran the Simulator a couple of times, and I achieved the Project Rubric as below:

- The car is able to drive at least 4.32 miles without incident.
- The car drives according to the speed limit.
- Max Acceleration and Jerk are not Exceeded.
- Car does not have collisions.
- The car stays in its lane, except for the time between changing lanes.
- The car is able to change lanes

If the left lane is not dense, the ego vehicle executes Lane Change to left.

If there's a car in the left lane and no car in the center lane, the ego vehicle executes Lane Change to center.

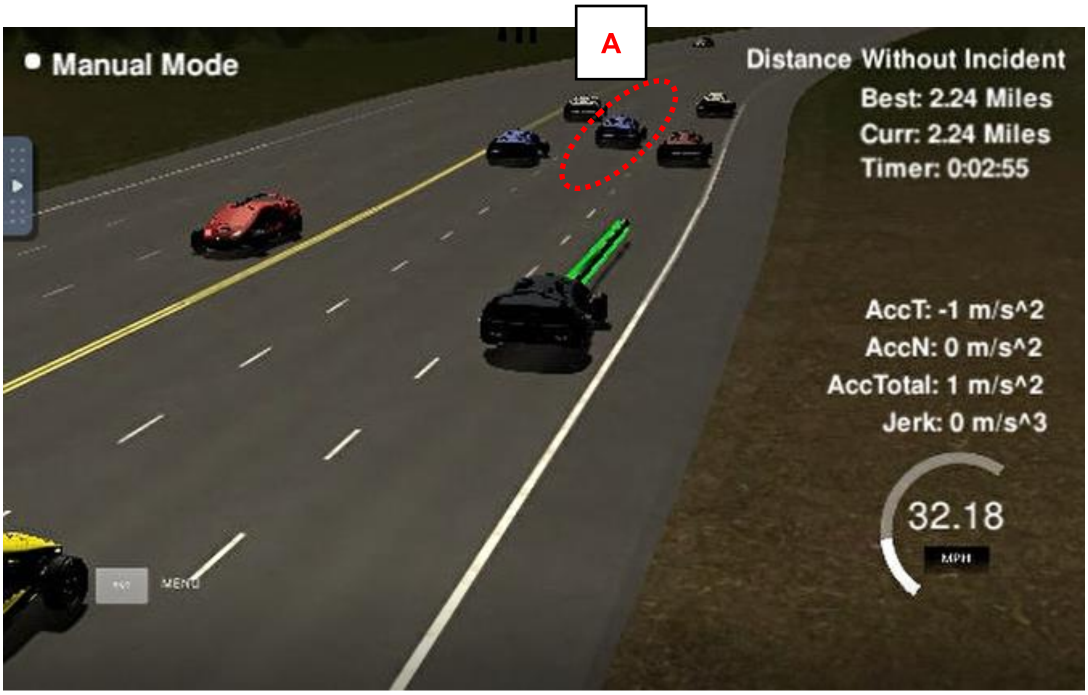


If there're cars in the left and center lanes and there's no car in the right lane, the ego vehicle executes Lane Change to right.

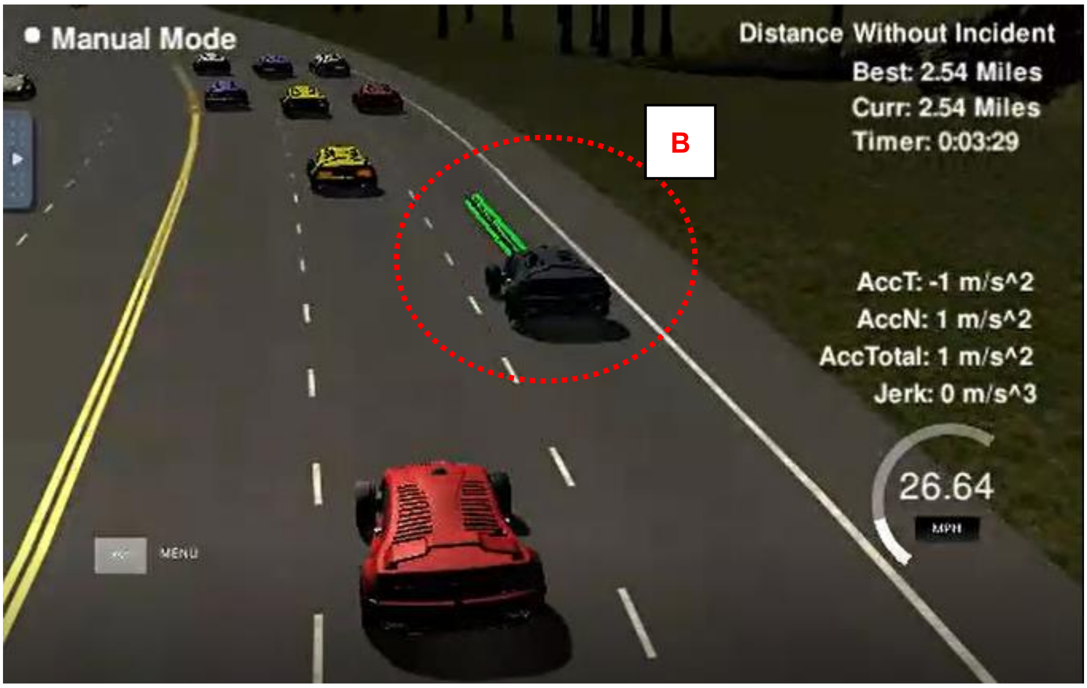
6. Future Improvement

If there's a traffic jam ahead, the ego vehicle cannot Lane Change and just keeps the current lane.

In the scene below, it was better to go to the center lane earlier when there was only one vehicle ahead. **(A)**



Finally the traffic jam gets worse and the ego vehicle stacks in the right lane. **(B)**



In the future, I may decrease Lane Change distance when the ego vehicle detects traffic jam because other vehicles' speeds are basically slower than usual at that scene.