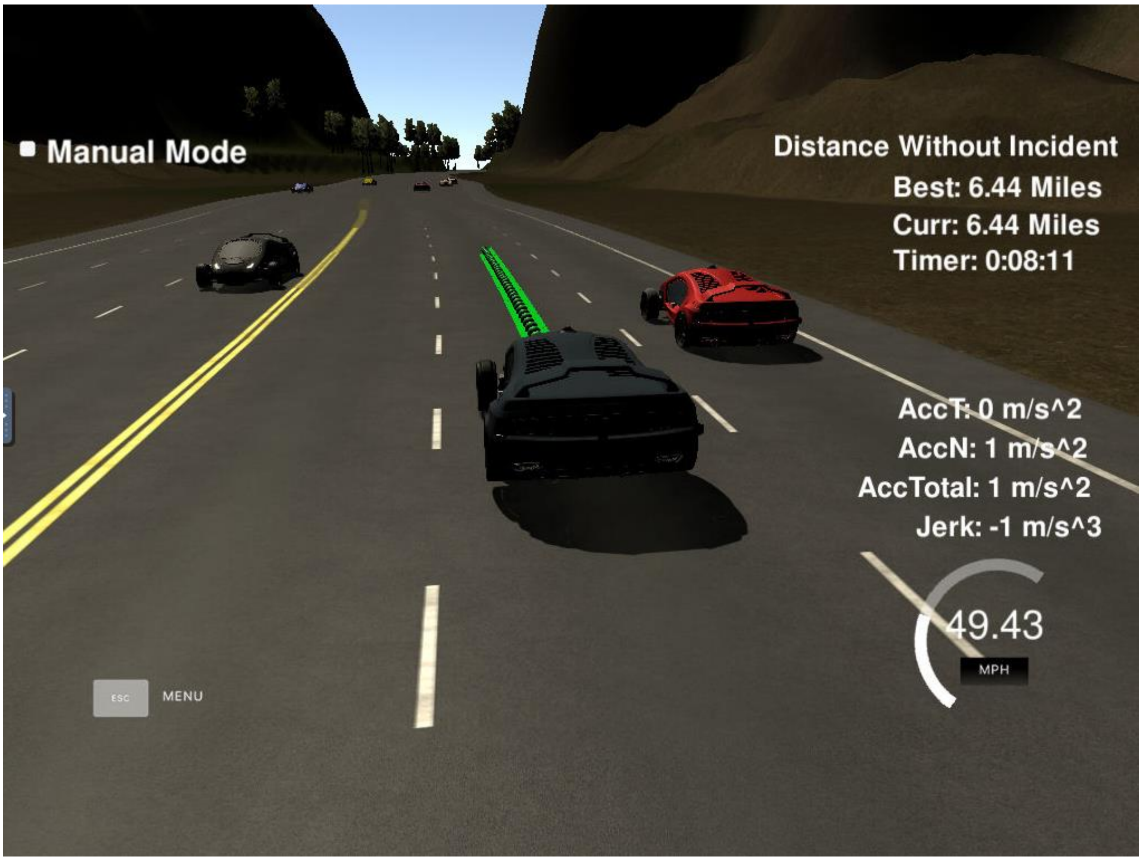
Path Planning Project

02/22/2019

Hakusho Chin



- Overview

This project is to navigate a self-driving car inside a highway.

The self-driving car should:

-Drive within a speed limit

-Max acceleration and jerk should not be exceeded

-The car does not collide with other cars

-the car stays in the lane, except for needed lane change

-the car is able to change lane

- Implementation

Decision Making

```
bool bIsTooClose = false;
bool bIsCarAtLeft = false;
bool bIsCarAtRight = false;

//check if other cars are in lane
for (int i = 0; i < sensor_fusion.size(); i++) {
    float d = sensor_fusion[i][6];
    int car_lane;
    if (d >= 0 && d < 4) {
        car_lane = 0;
    }
    else if (d >= 4 && d < 8) {
        car_lane = 1;
    }
    else if (d >= 8 && d <= 12) {
        car_lane = 2;
    }
    else {
        continue;
    }

    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_car_s += ((double)prev_size*0.02*check_speed);

    int gap = 30; // m
    if (car_lane == lane) {
        //there is a car ahead
        bIsTooClose |= (check_car_s > car_s) && ((check_car_s - car_s) < gap);
    }
    else if (car_lane - lane == 1) {
        //there is a car at right
        bIsCarAtRight |= ((car_s - gap) < check_car_s) && ((car_s + gap) > check_car_s);
    }
    else if (lane - car_lane == 1) {
        //there is a car at left
        bIsCarAtLeft |= ((car_s - gap) < check_car_s) && ((car_s + gap) > check_car_s);
    }
}
```

This part of code analyzes if there is a car ahead on your lane, or there is a car on your left or right. This information is processed from sensor fusion data.

```
//change lane condition
double acc = 0.224;
double max_speed = 49.5;
if (bIsTooClose) {
    // A car is ahead
    // Decide to shift lanes or slow down
    if (!bIsCarAtRight && lane < 2) {
        //if there is no car in the right, change lane to right
        lane++;
    }
    else if (!bIsCarAtLeft && lane > 0) {
        //if there is no car in the left, change lane to left
        lane--;
    }
    else {
        // no lanes available, deceleration
        ref_vel -= acc;
    }
}
else {
    if (lane != 1) {
        //return to center lane case
        if ((lane == 2 && !bIsCarAtLeft) || (lane == 0 && !bIsCarAtRight)) {
            // Move back to the center lane
            lane = 1;
        }
    }

    if (ref_vel < max_speed) {
        //no car in front, acceleration
        ref_vel += acc;
    }
}
```

This part of code shows policies regarding the ego vehicle's action, changing lane to right, left, or to accelerate, decelerate or go back to the center lane.

Trajectory Generation

```
vector<double> ptsx;
vector<double> ptsy;

// Reference x, y, yaw states
double ref_x = car_x;
double ref_y = car_y;
double ref_yaw = deg2rad(car_yaw);

//set reference
if (prev_size < 2) {
    double prev_car_x = car_x - cos(car_yaw);
    double prev_car_y = car_y - sin(car_yaw);

    ptsx.push_back(prev_car_x);
    ptsx.push_back(car_x);

    ptsy.push_back(prev_car_y);
    ptsy.push_back(car_y);
}
else {
    // Last point
    ref_x = previous_path_x[prev_size - 1];
    ref_y = previous_path_y[prev_size - 1];

    // 2nd-to-last point
    double ref_x_prev = previous_path_x[prev_size - 2];
    double ref_y_prev = previous_path_y[prev_size - 2];
    ref_yaw = atan2(ref_y - ref_y_prev, ref_x - ref_x_prev);

    ptsx.push_back(ref_x_prev);
    ptsx.push_back(ref_x);

    ptsy.push_back(ref_y_prev);
    ptsy.push_back(ref_y);
}

// create waypoints ahead
vector<double> next_wp0 = ConverttoXY(car_s + 30, (2 + 4 * lane), map_waypoints_s, map_waypoints_x, map_waypoints_y);
vector<double> next_wp1 = ConverttoXY(car_s + 60, (2 + 4 * lane), map_waypoints_s, map_waypoints_x, map_waypoints_y);
vector<double> next_wp2 = ConverttoXY(car_s + 90, (2 + 4 * lane), map_waypoints_s, map_waypoints_x, map_waypoints_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]);

for (int i = 0; i < ptsx.size(); i++) {
    //reset angle
    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));
}
```

Here we use the latest two points in the trajectory to calculate the waypoints ahead. We use spline to create trajectory for a planner, creating total of 50 waypoints.

```
// use spline to generate trajectory
tk::spline s;

// Set (x,y) points to the spline
s.set_points(ptsx, ptsy);

//trajectory sent to planner
vector<double> next_x_vals;
vector<double> next_y_vals;

for (int i = 0; i < previous_path_x.size(); i++) {
    next_x_vals.push_back(previous_path_x[i]);
    next_y_vals.push_back(previous_path_y[i]);
}

// Compute how to break up spline points so we travel at our desired reference velocity
double target_x = 30.0;
double target_y = s(target_x);
double target_dist = sqrt((target_x) * (target_x)+(target_y) * (target_y));
double x_add_on = 0;

// Fill up the rest of the path planner to always output 50 points
for (int i = 1; i <= 50 - previous_path_x.size(); i++) {
    double N = (target_dist / (.02*ref_vel / 2.24));
    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 earlier
    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_vals.push_back(x_point);
    next_y_vals.push_back(y_point);
}
```