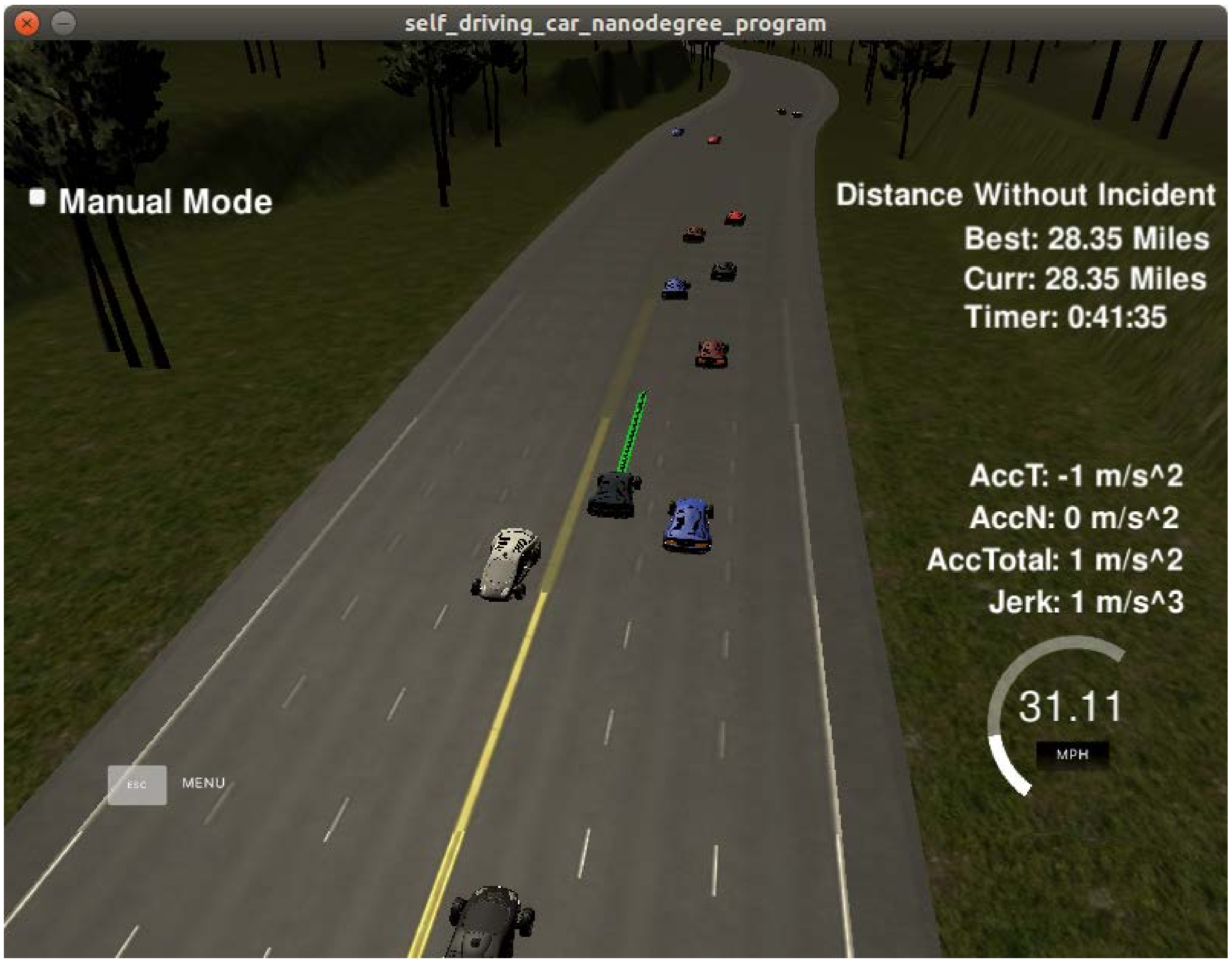
Path Planning Project



Reflections

My solution is based on the Jerk Minimization material as taught in the classroom and influenced by some ideas in the suggested paper [http://video.udacity-data.com.s3.amazonaws.com/topher/2017/July/595fd482_werling-optimal-trajectory-generation-for-dynamic-street-scenarios-in-a-frenet-frame/werling-optimal-trajectory-generation-for-dynamic-street-scenarios-in-a-frenet-frame.pdf].

I choose to use a different technique from that presented in the project walk-through, although with hindsight it may have been better to follow that approach. My solution is based around using the (s,d) Freenet co-ordinates for all computations and decision-making processes.

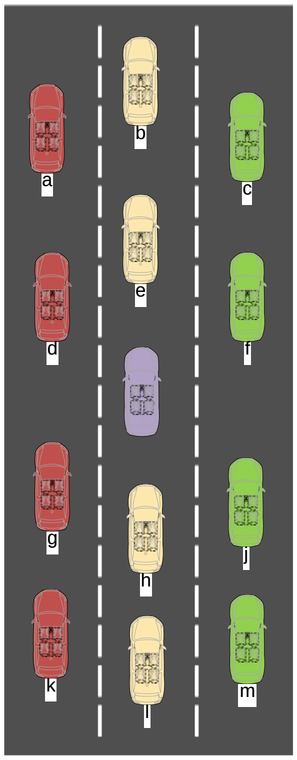
We refactored the provided code and added some extra files and classes. The refactored code places everything in a single PathData object.

We start by loading all the waypoints into our MapData object; we do this as although the provided Frenet to cartesian conversion algorithm given is technically correct - due to the simulator (which after all is a game engine geared towards x,y,z world) introduces discontinuities. To overcome this, we create four splines that given an s position returns the corresponding (x,y,nx,ny) values). It is not perfect either but does smooth out any sudden changes. The range of s is extended out by ten waypoints at each end to provide mappings at about [-300,7245] for reasons we will come to later.

The MapData class provides functions to convert from Freenet to Cartesian, and Cartesian to Frenet (need as sometimes the s, and d values are out by around +/- 16 m), along with s-position wrapping functions.

On each message from the simulator, we update the simulation time t_now (by the amount of previously used points x 20ms). We then convert the current car position and sensor fusion data, into VehicleData variables and pass to the "PathPlan:: TrackVehicle(int id, VehicleData v)" function which will create or get previous VehicleTracker object associated with the id given. At the same time, we update the distance of the vehicle to the main vehicle and maintain a grid of ids sorted by lane and distance, relative to our current lane. We can then get the vehicle id of the vehicle in front by GetNear(1, 2) or the vehicle in front of that by GetNear(1, 3) or the vehicle behind us in the adjacent right-hand lane to us by GetNear(2, 1).

	0	1	2
3	a	b	c
2	d	e	f
ego			
1	g	h	j
0	k	l	m



On each simulator update, the path calls UpdatePlan() , which in turn calls UpdateTarget() . UpdateTarget() is responsible for determining one of three modes either “Lane-changing”, “Following” or “Maximum Velocity” we should be doing. We use the very simple logic if we were in lane changing mode, and we reach our desired d-target we switch to one of the other modes. We the then check if any vehicle is in front of us and too close, and switch to follow mode and set the id of the vehicle to follow. We keep track of how long we are in follow mode, cycling through a range of keep distances – hoping that it frees up the possibility of changing lanes to a better one. If we are not changing lanes, we vote for how desirable the other lanes are, basically does it have enough space to do a change. If we get enough votes for a lane change, we score the current lane and the proposed lane. If the proposed lane has a bigger score, we target the new lane. Once UpdateTarget() has finished, UpdatePlan() will if necessary regenerate the s/d polynomials (because they have expired, or there is a new target).

We finally come to the heart of our path planning. The code assumes that at all times, we have generated a Quintic polynomial (if not necessarily optimal), such that we have a function for

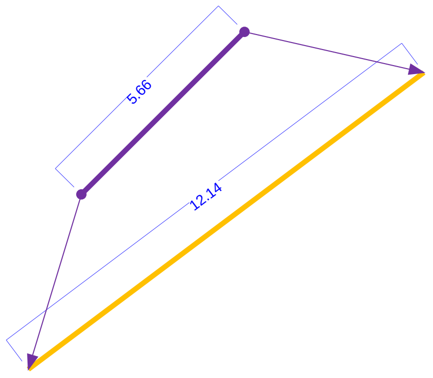
$$s(t) = a_5(t^5) + a_4(t^4) + a_3(t^3) + a_2(t^2) + a_1(t^1) + a_0$$

$$d(t) = b_5(t^5) + b_4(t^4) + b_3(t^3) + b_2(t^2) + b_1(t^1) + b_0$$

We can now very simple, repeat the previous five (if they exist) generated (x,y) pairs; We can then generate the remaining pairs by moving the current s,d position forward by s'(t).Δt and d'(t) Δt , and then working out the new x,y position. Which as s(t) is a polynomial we can easily calculate s'(t) =$5 * a_4 * t^4 + 4 * a_3 * t^3 + 3 * a_2 * t^2 + a_1 * t + a0$.

```
.
.
s_fwd = traj.s.get1d( t ) * 0.02 ;
d_fwd= traj.d.get1d( t ) * 0.02 ;
MoveForward( v_now , s_fwd , d_fwd ) ;
.
.
```

There is one problem, we have to adjust the s_fwd amount to compensate for how far off the centre we are in, as in this highly exaggerated example. The purple line has a real distance of 5.66m, and the yellow line a real distance of 12.14m, but they represent the same s-value. So we have to adjust our forward distance in this example by s_fwd * (5.66 / 12.14). These adjustments are taken care of in the MoveForward procedure.



The only remaining task is to generate the polynomials.

We generate the polynomials separately for both s and d.

The d polynomial tries to minimize $[\ d_0\ d_0'\ d_0''\ ] \rightarrow [\ d_1\ 0\ 0\ ]\ over\ 5\ seconds$.

The s-polynomial is a little harder. We, in essence, use a stochastic 'Montecarlo' like approach. We generate a random period Tp between [2.0,10.0] seconds, we then depending on the mode generate the ideal end conditions., and apply random pertubiations.

Velocitty following

$$s(t1) = s(t0) + Tp * 0.5 * (s'(t1) + MAX\_SPEED)$$
$$s'(t1) = MAX\_SPEED$$
$$s''(t1) = 0 .0$$

and when following a vehicle f .

$$s(t1) = f(0) + f'(0) * Tp – (K + 0.3 * f'(0))$$
$$s'(t1) = f'(t)$$
$$s''(t1) = 0.0$$

We then run several permutations of these end conditions, depending on the target mode. We then fit a jerk minimising trajectory quantic polynomial – and then check and score the resulting path. The checks rule out any paths that would exceed the speed/acceleration and jerk limits, and if acceptable gives them a score. We repeat the process with a new random period until we have found at least 50 good examples and use the trajectory with the best (lowest score).