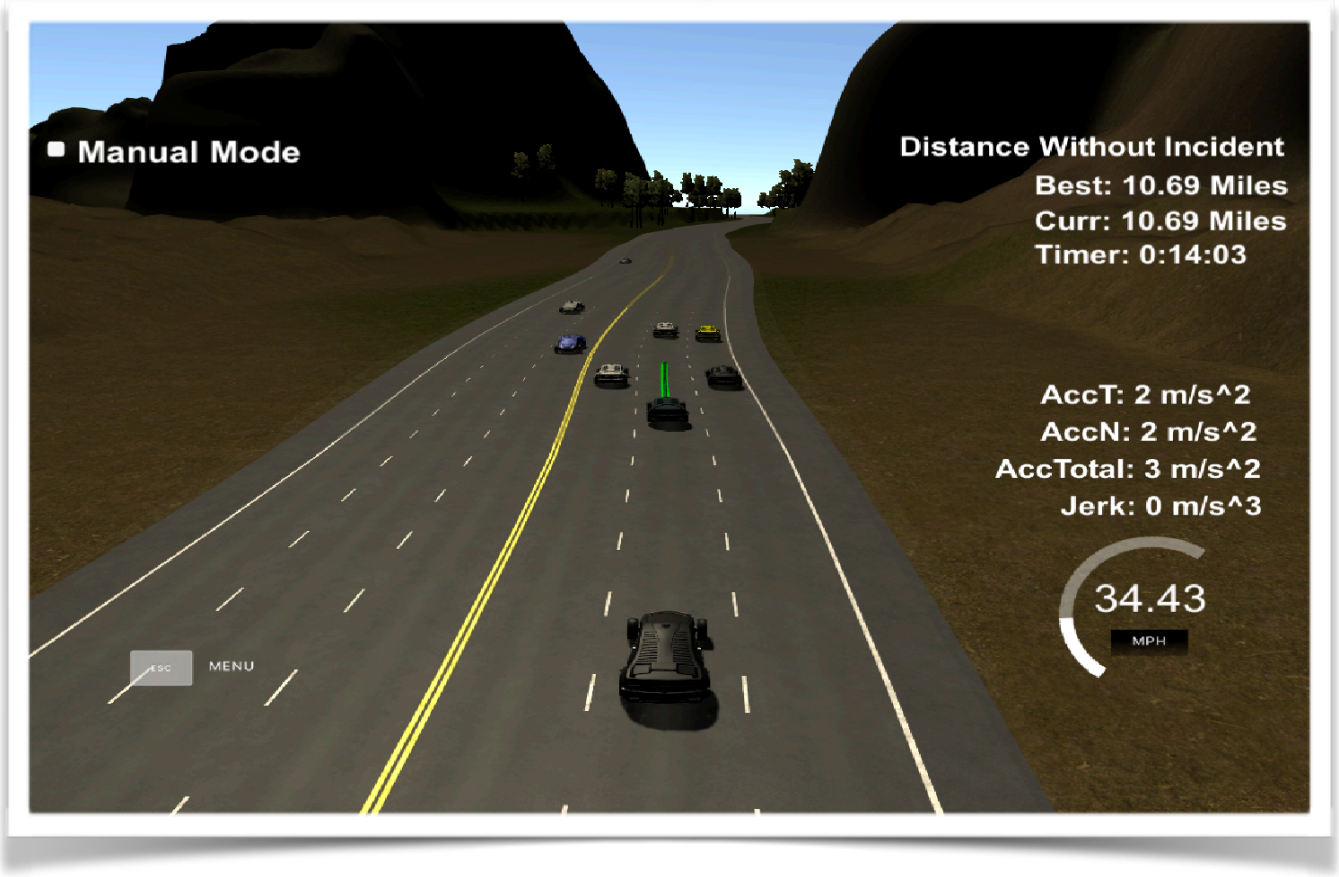
Welcome to Path Planning !!!



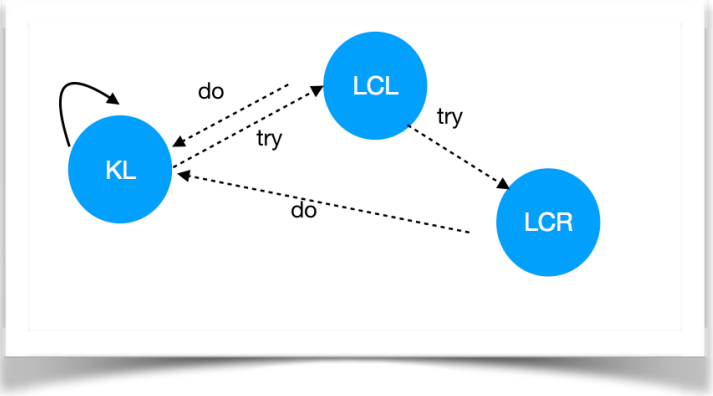
This project was developed two different ways :

- At first, I watched the Udacity's project walkthrough and followed the Finite State Machine implementation.
- Then, I tried a different way. This approach uses a Finite State Machine and Cost functions.

The two ways worked on my machine for at least 2 laps each.

First Way

The first way follows the implementation of the walkthrough.
It uses what I call a limited Finite State Machine.



Let’s go into the implementation.

```
We declare three variables corresponding to availabilities of lanes.
bool too_close = false;
bool left_free = true;
bool right_free = true;

We go through each sensor fusion data
for (int i=0; i<sensor_fusion.size();i++){
     double vx = sensor_fusion[i][3];
     double vy = sensor_fusion[i][4];
     double check_car_s = sensor_fusion[i][5];
     double d = sensor_fusion[i][6];
     double check_speed = sqrt(vx*vx+vy*vy);

Compute car s in the future
check_car_s +=((double)prev_size*.02*check_speed);

 If detected vehicle is in CAR lane & if a car is in our range – About 30 m

 if (d<(2+4*lane+2)&&(d>4*lane)){
 if ((check_car_s>car_s)&&((check_car_s-car_s) <30)){
             too_close=true; -> set STATE to : Current Lane Busy
    }
   }

// If detected vehicle in in LEFT LANE & If LEFT LANE is NOT FREE
  else if ((d<4*lane)&&(d>4*lane-4)){
    if(((check_car_s>car_s) && (check_car_s - car_s <30))||((check_car_s<car_s)&&(car_s-
check_car_s <10 )))
            {
              left_free = false; -> set STATE to : Left Busy
            }
          }
// If detected vehicle in in RIGHT LANE & if RIGHT LANE is NOT FREE

          else if ((d<4*lane+8)&&(d>4*lane+4)){
           if(((check_car_s>car_s) && (check_car_s - car_s <30))||
((check_car_s<car_s)&&(car_s-check_car_s <10 )))
            {
                right_free = false; -> set STATE to : Right Busy
            }
          }
        }

Then, if we’re too close, we check possibilities : Left or Right. We always give
advantage to LEFT LANE for going over.

if(too_close){
     if((left_free)&&(lane>0)){
            lane-=1; -> Decrease Lane
          }
      else if ((right_free)&&(lane<2))lane+=1; -> Increase Lane
          else {
            ref_vel-=.224; -> Decrease Speed
          }
        }
        else if (ref_vel<49.5){
          ref_vel+=.224; -> Increase Speed
        }
```

If there is a car in front of us, we go left is free. If not, we go right. If not, we decrease speed. If our lane if free, we go up to speed limit. We add .224 mph because if we derivate this with time of the simulator 0.02, we have a jerk of 10 m.s-2; which is maximum jerk of comfort allowed.

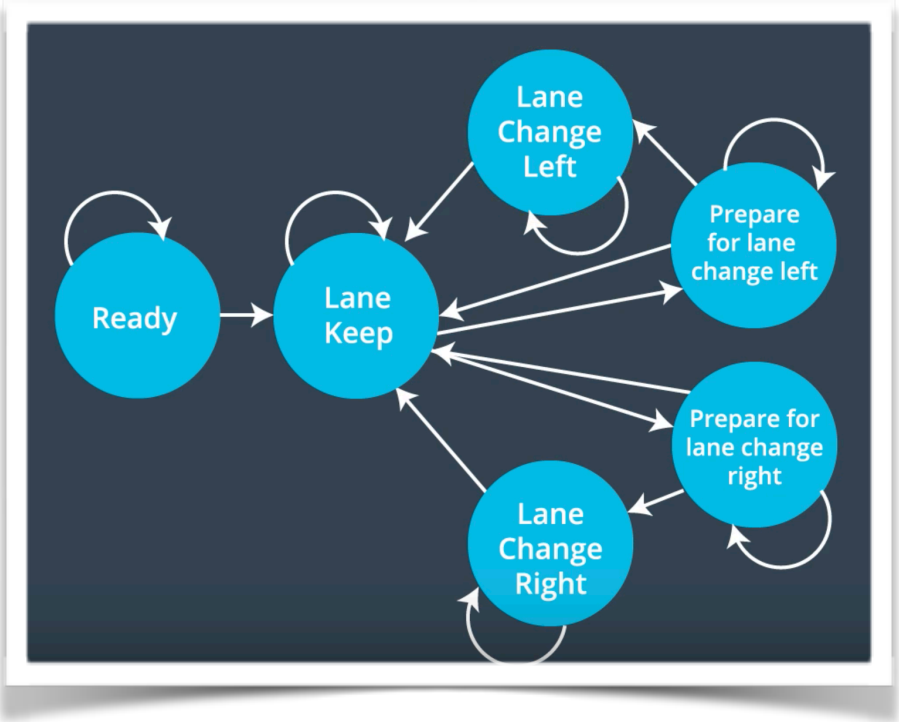
Results

- This implementation has good performance. It is simple and not too hard to code. Everything can be done in main.cpp without any class or object.
- If a car suddenly changes lane and run into me, the car will not really predict and will wait until the car is completely in lane to slow down. This might do collisions.
- The going over algorithm is « handcoded ».
- If a car in the left lane is slow, we might be stuck behind a long time if right lane isn't free.

Second Way

The second way uses a behavioral planner from Udacity's class.

This implementation uses a different FSM I will comment.



In this case, we don't always lane keep unless a vehicle is in front of us.

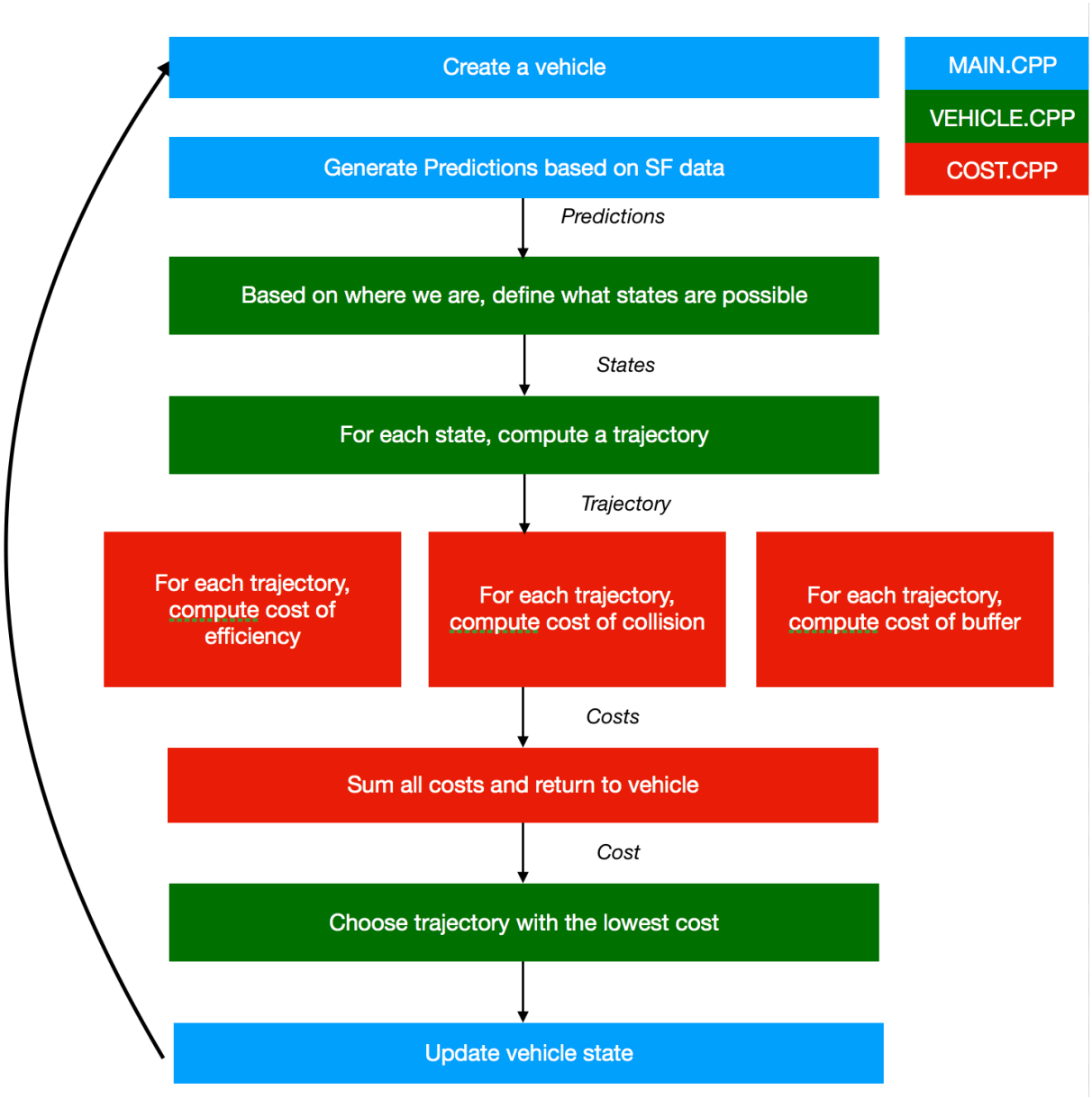
At each loop, we have three states :

- Lane Keeping
- Prepare for change left
- Prepare for change right

The trajectory for these three states is the same. Except that **prepare for change checks if there is a way of changing lane.** If so, we go into that state if it's better for us.

What does better for us mean ?

It means that for each trajectory, we compute a cost based on questions. *Is our lane slow ? Are other lanes really free ? Are others lanes faster ?*
Then, we select the lowest cost.



Results

- This implementation is more sophisticated and can avoid being stucked behind a slow car by computing the more efficient trajectory.
- The work to implement it was way harder and longer than the first one. It took several days to get something still colliding sometimes.
- It will not always go over a slow car if it estimates that the other lanes are not faster. Which seems good in theory but not really in practice since we all change lanes even if they are not free.
- It computes lane by lane trajectory, and doesn't look 2 lanes further which is not optimum.
- The prepare lane change state is somehow not efficient in a way that our cost function might choose to Keep Lane instead of changing lane if all costs are 0.

Once the vehicle has a lane and a velocity to go, it generates a spline based on 3 points spaced by 30 meters. This splines makes sure to go through every point of the trajectory and is better than polynomial fit for this reason.

Notes to the reviewer :
I guarantee the first implementation to work.
I tried something more sophisticated for the second implementation and it was a real struggle to get the car going somewhere.
I do not guarantee the second implementation to work perfectly and you might even encounter jerk issues. I tried to apply a lane change cost to avoid lateral jerk but it did get the other behaviors wrong.
I however could go over 10 miles with both implementations without staying in my lane and with going accordingly to traffic speed.