Path Planning for Autonomous Vehicles

This project implements a path planning algorithm for a self-driving car in a simulated highway environment. The system is designed to navigate the car safely and efficiently while adhering to speed limits, avoiding collisions, and making appropriate lane changes. Here's a detailed technical explanation of the project:

Core Components

1. **Main Program (main.cpp)**
   - Initializes the environment and manages the main loop of the program.
   - Handles communication with the simulator using WebSocket.
   - Processes incoming data and generates trajectory points.

2. **Vehicle Class (vehicle.h, vehicle.cpp)**
   - Represents the ego vehicle and other vehicles on the road.
   - Manages vehicle state and behavior.

3. **Helper Functions (helpers.h)**
   - Provides utility functions for coordinate transformations and calculations.

Key Algorithms and Techniques

1. Sensor Fusion and Environment Perception

The program processes sensor fusion data to understand the environment:
cpp
```cpp
for (int i = 0; i < sensor_fusion.size(); i++) {
  // Extract data for each detected vehicle
  // Create Vehicle objects for other cars
  Vehicle car(other_car_lane, other_car_s, other_car_speed);
  ego_car.AssessOtherCar(car);
}
```

This loop analyzes the position, speed, and lane of other vehicles relative to the ego car.

2. Behavior Planning

The `ChooseNextMove` function in the Vehicle class implements the decision-making process:
cpp
```cpp
double Vehicle::ChooseNextMove(double ref_vel) {
  float cost_left = this->CalculateCost("CLL");
  float cost_right = this->CalculateCost("CLR");
  float cost_keep = this->CalculateCost("KL");
  // Choose the action with the lowest cost
}
```

This function evaluates three possible actions: changing lane left, changing lane right, or keeping the current lane. It uses a cost function to determine the best action.

3. Trajectory Generation

The main program uses a spline-based approach to generate smooth trajectories:

1. It creates anchor points using the current position and future waypoints.
2. These points are transformed to the car's local coordinate system.
3. A spline is fitted to these points:

cpp

```
tk::spline s;
s.set_points(ptsx, ptsy);
```

4. The spline is then used to generate a smooth path of 50 points:

cpp

```
for (int i = 0; i <= 50-previous_path_x.size(); i++) {
  // Calculate point along spline
  // Transform back to global coordinates
  next_x_vals.push_back(x_point);
  next_y_vals.push_back(y_point);
}
```

4. Speed Control

The program implements gradual acceleration and deceleration to ensure passenger comfort:

cpp

```
double Vehicle::KeepLane(double ref_vel) {
  if (this->car_ahead) {
    this->acceleration = -this->MAX_ACC;
  } else if (ref_vel < this->MAX_SPEED) {
    this->acceleration = this->MAX_ACC;
  }
  ref_vel += this->acceleration;
  // Ensure speed doesn't exceed maximum
}
```

5. Coordinate Transformations

The project uses both Cartesian (x, y) and Frenet (s, d) coordinate systems. Helper functions in `helpers.h` provide transformations between these systems:

- `getFrenet`: Converts from Cartesian to Frenet coordinates.
- `getXY`: Converts from Frenet to Cartesian coordinates.

These transformations are crucial for lane-based reasoning and trajectory generation.

Key Features

1. **Lane Changing**: The car can change lanes when it's safe and beneficial.
2. **Speed Adaptation**: The car adjusts its speed based on traffic conditions.
3. **Collision Avoidance**: The system maintains safe distances from other vehicles.
4. **Smooth Trajectories**: Using spline interpolation ensures smooth and comfortable paths.
5. **Efficient Path Planning**: The algorithm aims to keep the car moving at the speed limit when safe.

Challenges and Considerations

1. **Safety**: Ensuring safe distances and maneuvers in all scenarios.
2. **Efficiency**: Balancing between reaching the speed limit and making unnecessary lane changes.
3. **Comfort**: Limiting acceleration and jerk for passenger comfort.
4. **Prediction**: Anticipating the behavior of other vehicles for better decision-making.

This project demonstrates a comprehensive approach to autonomous vehicle path planning, incorporating perception, decision making, and control aspects of self-driving technology.

Code:

Main.cpp

```
#include <uWS/uWS.h>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "Eigen-3.3/Eigen/Core"
#include "Eigen-3.3/Eigen/QR"
#include "helpers.h"
#include "json.hpp"
#include "spline.h"
#include "vehicle.h"

using nlohmann::json;
using std::string;
using std::vector;

int main() {
 uWS::Hub h;

 // Load up map values for waypoint's x,y,s and d normalized normal
vectors
 vector<double> map_waypoints_x;
 vector<double> map_waypoints_y;
 vector<double> map_waypoints_s;
 vector<double> map_waypoints_dx;
 vector<double> map_waypoints_dy;

 // Waypoint map to read from
 string map_file_ = "../data/highway_map.csv";

 std::ifstream in_map_(map_file_.c_str(), std::ifstream::in);

 string line;
 while (getline(in_map_, line)) {
```

```
    std::istringstream iss(line);
    double x;
    double y;
    float s;
    float d_x;
    float d_y;
    iss >> x;
    iss >> y;
    iss >> s;
    iss >> d_x;
    iss >> d_y;
    map_waypoints_x.push_back(x);
    map_waypoints_y.push_back(y);
    map_waypoints_s.push_back(s);
    map_waypoints_dx.push_back(d_x);
    map_waypoints_dy.push_back(d_y);
  }

  // start in lane 1
  int lane = 1;

  // reference velocity target (unit: mph)
  double ref_vel = 0.0;

  h.onMessage([&map_waypoints_x,&map_waypoints_y,&map_waypoints_s,
               &map_waypoints_dx,&map_waypoints_dy, &lane, &ref_vel]
              (uWS::WebSocket<uWS::SERVER> ws, char *data, size_t length,
               uWS::OpCode opCode) {
    // "42" at the start of the message means there's a websocket message
event.
    // The 4 signifies a websocket message
    // The 2 signifies a websocket event
    if (length && length > 2 && data[0] == '4' && data[1] == '2') {

      auto s = hasData(data);

      if (s != "") {
        auto j = json::parse(s);

        string event = j[0].get<string>();
```

```
        if (event == "telemetry") {
          // j[1] is the data JSON object

          // Main car's localization Data
          double car_x = j[1]["x"];
          double car_y = j[1]["y"];
          double car_s = j[1]["s"];
          double car_d = j[1]["d"];
          double car_yaw = j[1]["yaw"];
          double car_speed = j[1]["speed"];

          // Previous path data given to the Planner
          auto previous_path_x = j[1]["previous_path_x"];
          auto previous_path_y = j[1]["previous_path_y"];

          // Previous path's end s and d values
          double end_path_s = j[1]["end_path_s"];
          double end_path_d = j[1]["end_path_d"];

          // Sensor Fusion Data, a list of all other cars on the same side
of the road.
          auto sensor_fusion = j[1]["sensor_fusion"];

          json msgJson;

          int prev_size = previous_path_x.size();

          if (prev_size > 0)
          {
            car_s = end_path_s;
          }

          // create a vehicle object for the ego car
          Vehicle ego_car(lane, car_s, car_speed);

          // loop over other cars
          for (int i = 0; i < sensor_fusion.size(); i++){
            double other_car_vx = sensor_fusion[i][3];
            double other_car_vy = sensor_fusion[i][4];
```

```
            double other_car_s = sensor_fusion[i][5];
            float other_car_d = sensor_fusion[i][6];

            // convert frenet coordinate d to lane number (assumption: lane width is 4 meters)
            int other_car_lane = GetLane(other_car_d,4);

            double other_car_speed = sqrt(other_car_vx*other_car_vx + other_car_vy*other_car_vy);

            // position after executing previous trajectory
            other_car_s += ((double)prev_size * 0.02 * other_car_speed);

            // create a vehicle object for the other car
            Vehicle car(other_car_lane, other_car_s, other_car_speed);

            // analyze other car (it's speed, distance, lane) with resect to ego car to help with path planning
            ego_car.AssessOtherCar(car);

          }

          // decide on the next best move: find velocity and lane
          ref_vel = ego_car.ChooseNextMove(ref_vel);
          lane = ego_car.lane;

          // create a list of widely and evenly spaced waypoints at 30m
          vector<double> ptsx;
          vector<double> ptsy;

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

          // use the car location as starting 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);
        }
        // use the previous path's end points as starting reference
        else
        {
          // set reference as previous path end point
          ref_x = previous_path_x[prev_size - 1];
          ref_y = previous_path_y[prev_size - 1];

          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);

          // use two points that make the path tangent to the previous
path's end point
          ptsx.push_back(ref_x_prev);
          ptsx.push_back(ref_x);

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

        // add evenly spaced points (30m) ahead of the starting reference
        vector<double> next_wp0 = getXY(car_s+30, 2+4*lane,
map_waypoints_s, map_waypoints_x, map_waypoints_y);
        vector<double> next_wp1 = getXY(car_s+60, 2+4*lane,
map_waypoints_s, map_waypoints_x, map_waypoints_y);
        vector<double> next_wp2 = getXY(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) {
  // shift car reference angle to 0
  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
tk::spline s;

s.set_points(ptsx, ptsy);

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]);
}

// break up spline points for travelling at 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
for (int i = 0; i <= 50-previous_path_x.size(); i++)
{
  double N = target_dist / (0.02*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;

          // transformation
          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);

        }

        msgJson["next_x"] = next_x_vals;
        msgJson["next_y"] = next_y_vals;

        auto msg = "42[\"control\","+ msgJson.dump()+"]";

        ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
      }  // end "telemetry" if
    } else {
      // Manual driving
      std::string msg = "42[\"manual\",{}]";
      ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
    }
  }  // end websocket if
}); // end h.onMessage

h.onConnection([&h](uWS::WebSocket<uWS::SERVER> ws, uWS::HttpRequest req)
{
  std::cout << "Connected!!!" << std::endl;
});

h.onDisconnection([&h](uWS::WebSocket<uWS::SERVER> ws, int code,
                       char *message, size_t length) {
  ws.close();
```

```
    std::cout << "Disconnected" << std::endl;
  });

  int port = 4567;
  if (h.listen(port)) {
    std::cout << "Listening to port " << port << std::endl;
  } else {
    std::cerr << "Failed to listen to port" << std::endl;
    return -1;
  }

  h.run();
}
```

Vehicle.h

```
#ifndef VEHICLE_H
#define VEHICLE_H

#include <string>
#include <vector>

using std::string;
using std::vector;

class Vehicle {
public:
  // Constructors
  Vehicle();
  Vehicle(int lane, float s, float speed);

  // Destructor
  virtual ~Vehicle();

  void AssessOtherCar(Vehicle car);
  double ChooseNextMove(double ref_vel);
  float CalculateCost(string state);
  double KeepLane(double ref_vela);

  const int MIN_SAFE_DISTANCE = 30;
  const double MAX_SPEED = 49.5;
```

```
 const double MAX_ACC = 0.224;

 int lane;
 float s, d, speed, acceleration;
 bool car_ahead, car_right, car_left;
 float car_ahead_speed, car_ahead_distance;
 float car_left_speed, car_left_distance;
 float car_right_speed, car_right_distance;
};

#endif  // VEHICLE_H
```

Vehicle.cpp

```
#include "vehicle.h"
#include <map>
#include <string>
#include <vector>

#include <iostream>
using namespace std;
using std::string;
using std::vector;
using std::min;


// Initializes Vehicle
Vehicle::Vehicle(){}

Vehicle::Vehicle(int lane, float s, float speed) {
 this->lane = lane;
 this->s = s;
 this->speed = speed;
 this->acceleration = 0;
 this->car_ahead = false;
 this->car_left = false;
 this->car_right = false;
 this->car_ahead_speed = 0;
 this->car_ahead_distance = 10000;
```

```
 this->car_left_speed = 0;
 this->car_left_distance = 10000;
 this->car_right_speed = 0;
 this->car_right_distance = 10000;
}

Vehicle::~Vehicle() {}

void Vehicle::AssessOtherCar(Vehicle car) {
 float distance = car.s - this->s;

 if (car.lane == this->lane) {
   // check if there is a car ahead
   this->car_ahead |= (car.s > this->s) && (car.s - this->s <
car.MIN_SAFE_DISTANCE);
   // record the distnace and speed of the car ahead in this lane
   if ((distance < this->car_ahead_distance) && (distance > 0)) {
     this->car_ahead_speed = car.speed;
     this->car_ahead_distance = distance;
   }
 } else if (car.lane == this->lane - 1){
   // check if there is a car in the left lane
   this->car_left |= (distance < car.MIN_SAFE_DISTANCE) && (distance >
-car.MIN_SAFE_DISTANCE);
   // record the distnace and speed of the car in the left lane
   if ((distance < this->car_left_distance) && (distance > 0)) {
     this->car_left_speed = car.speed;
     this->car_left_distance = distance;
   }
 } else if (car.lane = this->lane + 1) {
   // check if there is a car in the right lane
   this->car_right |= (distance < car.MIN_SAFE_DISTANCE) && (distance >
-car.MIN_SAFE_DISTANCE);
   // record the distnace and speed of the car in the right lane
   if ((distance < this->car_right_distance) && (distance > 0)) {
     this->car_right_speed = car.speed;
     this->car_right_distance = distance;
   }
 }
}
```

```cpp
double Vehicle::ChooseNextMove(double ref_vel) {

 // calculate the cost of three possible moves:
 //    1. Change Lane Left (CLL)
 //    2. Change Lane Right (CLR)
 //    3. Keep Lane (KL)
 float cost_left = this->CalculateCost("CLL");
 float cost_right = this->CalculateCost("CLR");
 float cost_keep = this->CalculateCost("KL");

 float cost_min = min(cost_left, min(cost_right,cost_keep));

 // path planning based on mimizing cost
 if (cost_keep == cost_min) {
   ref_vel = this->KeepLane(ref_vel);
 } else if (cost_right == cost_min) {
   this->lane++;
 } else {
   this->lane--;
 }
 return ref_vel;
}


double Vehicle::KeepLane(double ref_vel) {
 if (this->car_ahead) {
   // slow down if there is a car ahead
   this->acceleration = -this->MAX_ACC;
 } else {
   if (ref_vel < this->MAX_SPEED) {
     // speed up until speed reaches to max, there is no car ahead
     this->acceleration = this->MAX_ACC;
   }
 }
 ref_vel += this->acceleration;

 if (ref_vel > this->MAX_SPEED) {
   ref_vel = this->MAX_SPEED;
 }
```

```
 return ref_vel;
}


float Vehicle::CalculateCost(string state) {
 float lane_speed = 0;

 if (state == "CLL") {
   if ((this->lane == 0) || this->car_left || !this->car_ahead) {
     return 1;
   } else {
     lane_speed = this->car_left_speed;
   }
 }

 if (state == "CLR") {
   if ((this->lane == 2) || this->car_right || !this->car_ahead) {
     return 1;
   } else {
     lane_speed = this->car_right_speed;
   }
 }

 if (state == "KL") {
   if (car_ahead) return 1;
 } else {
   lane_speed = this->car_ahead_speed;
 }

 // calculate cost function based on speed of the lane, cost is between 0
and 1
 float cost = (this->MAX_SPEED - lane_speed)/this->MAX_SPEED;
 return cost;

}
```

Helpers.h code

```
#ifndef HELPERS_H
#define HELPERS_H
```

```
#include <math.h>
#include <string>
#include <vector>

// for convenience
using std::string;
using std::vector;

// Checks if the SocketIO event has JSON data.
// If there is data the JSON object in string format will be returned,
//   else the empty string "" will be returned.
string hasData(string s) {
 auto found_null = s.find("null");
 auto b1 = s.find_first_of("[");
 auto b2 = s.find_first_of("}");
 if (found_null != string::npos) {
   return "";
 } else if (b1 != string::npos && b2 != string::npos) {
   return s.substr(b1, b2 - b1 + 2);
 }
 return "";
}

//
// Helper functions related to waypoints and converting from XY to Frenet
//   or vice versa
//

// For converting back and forth between radians and degrees.
constexpr double pi() { return M_PI; }
double deg2rad(double x) { return x * pi() / 180; }
double rad2deg(double x) { return x * 180 / pi(); }

// Calculate distance between two points
double distance(double x1, double y1, double x2, double y2) {
 return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}

// Calculate closest waypoint to current x, y position
```

```
int ClosestWaypoint(double x, double y, const vector<double> &maps_x,
                    const vector<double> &maps_y) {
 double closestLen = 100000; //large number
 int closestWaypoint = 0;

 for (int i = 0; i < maps_x.size(); ++i) {
   double map_x = maps_x[i];
   double map_y = maps_y[i];
   double dist = distance(x,y,map_x,map_y);
   if (dist < closestLen) {
     closestLen = dist;
     closestWaypoint = i;
   }
 }

 return closestWaypoint;
}

// Returns next waypoint of the closest waypoint
int NextWaypoint(double x, double y, double theta, const vector<double>
&maps_x,
                const vector<double> &maps_y) {
 int closestWaypoint = ClosestWaypoint(x,y,maps_x,maps_y);

 double map_x = maps_x[closestWaypoint];
 double map_y = maps_y[closestWaypoint];

 double heading = atan2((map_y-y),(map_x-x));

 double angle = fabs(theta-heading);
 angle = std::min(2*pi() - angle, angle);

 if (angle > pi()/2) {
   ++closestWaypoint;
   if (closestWaypoint == maps_x.size()) {
     closestWaypoint = 0;
   }
 }

 return closestWaypoint;
```

```
}

// Transform from Cartesian x,y coordinates to Frenet s,d coordinates
vector<double> getFrenet(double x, double y, double theta,
                         const vector<double> &maps_x,
                         const vector<double> &maps_y) {
 int next_wp = NextWaypoint(x,y, theta, maps_x,maps_y);

 int prev_wp;
 prev_wp = next_wp-1;
 if (next_wp == 0) {
   prev_wp  = maps_x.size()-1;
 }

 double n_x = maps_x[next_wp]-maps_x[prev_wp];
 double n_y = maps_y[next_wp]-maps_y[prev_wp];
 double x_x = x - maps_x[prev_wp];
 double x_y = y - maps_y[prev_wp];

 // find the projection of x onto n
 double proj_norm = (x_x*n_x+x_y*n_y)/(n_x*n_x+n_y*n_y);
 double proj_x = proj_norm*n_x;
 double proj_y = proj_norm*n_y;

 double frenet_d = distance(x_x,x_y,proj_x,proj_y);

 //see if d value is positive or negative by comparing it to a center
point
 double center_x = 1000-maps_x[prev_wp];
 double center_y = 2000-maps_y[prev_wp];
 double centerToPos = distance(center_x,center_y,x_x,x_y);
 double centerToRef = distance(center_x,center_y,proj_x,proj_y);

 if (centerToPos <= centerToRef) {
   frenet_d *= -1;
 }

 // calculate s value
 double frenet_s = 0;
 for (int i = 0; i < prev_wp; ++i) {
```

```cpp
    frenet_s += distance(maps_x[i],maps_y[i],maps_x[i+1],maps_y[i+1]);
  }

  frenet_s += distance(0,0,proj_x,proj_y);

  return {frenet_s,frenet_d};
}

// Transform from Frenet s,d coordinates to Cartesian x,y
vector<double> getXY(double s, double d, const vector<double> &maps_s,
                     const vector<double> &maps_x,
                     const vector<double> &maps_y) {
  int prev_wp = -1;

  while (s > maps_s[prev_wp+1] && (prev_wp < (int)(maps_s.size()-1))) {
    ++prev_wp;
  }

  int wp2 = (prev_wp+1)%maps_x.size();

  double heading = atan2((maps_y[wp2]-maps_y[prev_wp]),
                         (maps_x[wp2]-maps_x[prev_wp]));
  // the x,y,s along the segment
  double seg_s = (s-maps_s[prev_wp]);

  double seg_x = maps_x[prev_wp]+seg_s*cos(heading);
  double seg_y = maps_y[prev_wp]+seg_s*sin(heading);

  double perp_heading = heading-pi()/2;

  double x = seg_x + d*cos(perp_heading);
  double y = seg_y + d*sin(perp_heading);

  return {x,y};
}

// Transform from Frenet d coordinate to lane number
int GetLane(float d, float lane_width) {
  // get the lane of a car from it's distance from the middle of the road
  int lane = (int)d/lane_width;
```

```
 if (lane < 0) {
     return -1;
 } else {
   return lane;
 }
}

#endif  // HELPERS_H
```

Result