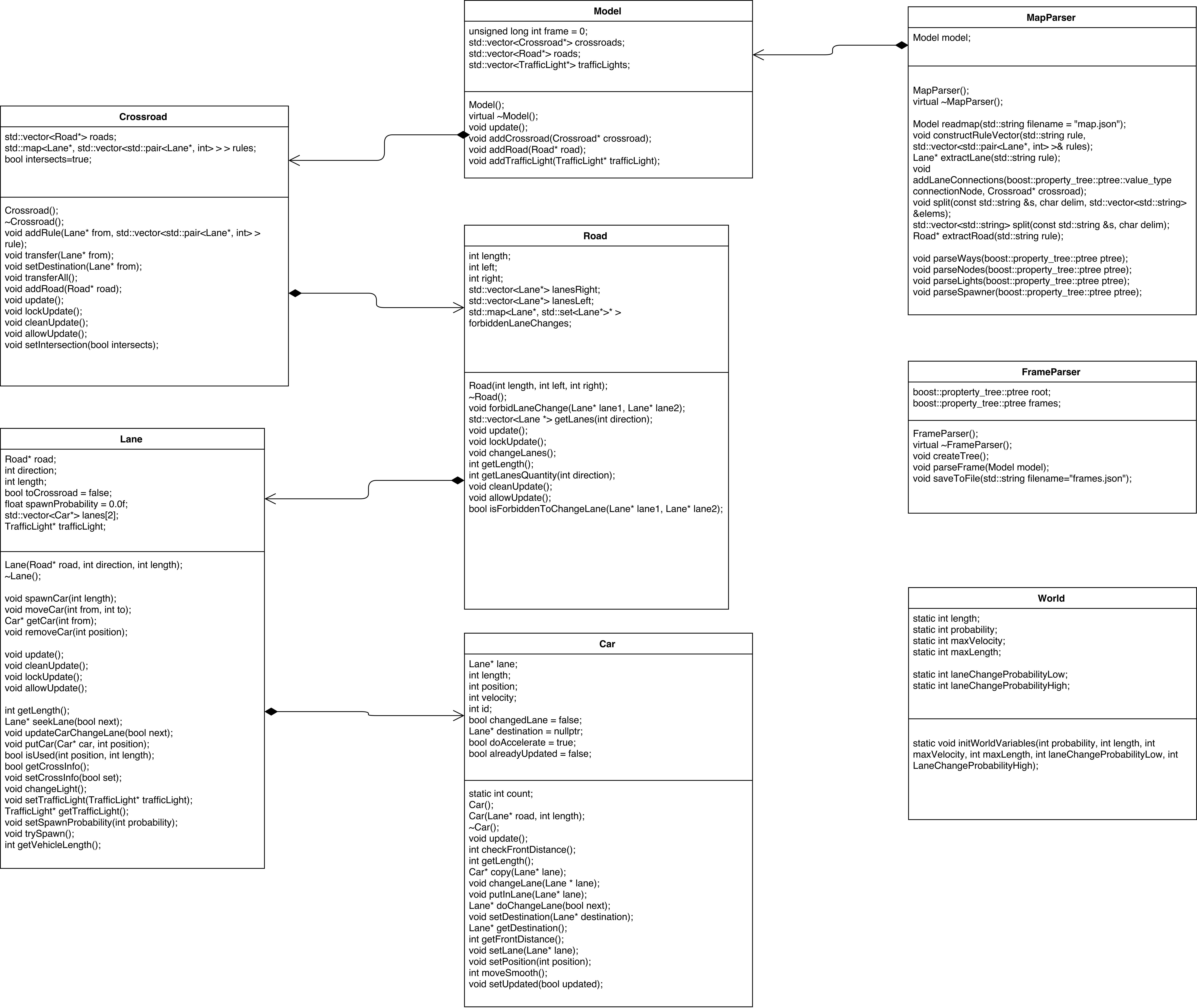

Model

```
unsigned long int frame = 0;
std::vector<Crossroad*> crossroads;
std::vector<Road*> roads;
std::vector<TrafficLight*> trafficLights;
```

```
Model();
virtual ~Model();
void update();
void addCrossroad(Crossroad* crossroad);
void addRoad(Road* road);
void addTrafficLight(TrafficLight* trafficLight);
```

MapParser

```
Model model;
```

```
MapParser();
virtual ~MapParser();

Model readmap(std::string filename = "map.json");
void constructRuleVector(std::string rule,
std::vector<std::pair<Lane*, int> >& rules);
Lane* extractLane(std::string rule);
void
addLaneConnections(boost::property_tree::ptree::value_type
connectionNode, Crossroad* crossroad);
void split(const std::string &s, char delim, std::vector<std::string>
&elems);
std::vector<std::string> split(const std::string &s, char delim);
Road* extractRoad(std::string rule);

void parseWays(boost::property_tree::ptree ptree);
void parseNodes(boost::property_tree::ptree ptree);
void parseLights(boost::property_tree::ptree ptree);
void parseSpawner(boost::property_tree::ptree ptree);
```

Crossroad

```
std::vector<Road*> roads;
std::map<Lane*, std::vector<std::pair<Lane*, int> > > rules;
bool intersects=true;
```

```
Crossroad();
~Crossroad();
void addRule(Lane* from, std::vector<std::pair<Lane*, int> >
rule);
void transfer(Lane* from);
void setDestination(Lane* from);
void transferAll();
void addRoad(Road* road);
void update();
void lockUpdate();
void cleanUpdate();
void allowUpdate();
void setIntersection(bool intersects);
```

Road

```
int length;
int left;
int right;
std::vector<Lane*> lanesRight;
std::vector<Lane*> lanesLeft;
std::map<Lane*, std::set<Lane*>* >
forbiddenLaneChanges;
```

```
Road(int length, int left, int right);
~Road();
void forbidLaneChange(Lane* lane1, Lane* lane2);
std::vector<Lane *> getLanes(int direction);
void update();
void lockUpdate();
void changeLanes();
int getLength();
int getLanesQuantity(int direction);
void cleanUpdate();
void allowUpdate();
bool isForbiddenToChangeLane(Lane* lane1, Lane* lane2);
```

FrameParser

```
boost::propterty_tree::ptree root;
boost::property_tree::ptree frames;
```

```
FrameParser();
virtual ~FrameParser();
void createTree();
void parseFrame(Model model);
void saveToFile(std::string filename="frames.json");
```

Lane

```
Road* road;
int direction;
int length;
bool toCrossroad = false;
float spawnProbability = 0.0f;
std::vector<Car*> lanes[2];
TrafficLight* trafficLight;
```

```
Lane(Road* road, int direction, int length);
~Lane();

void spawnCar(int length);
void moveCar(int from, int to);
Car* getCar(int from);
void removeCar(int position);

void update();
void cleanUpdate();
void lockUpdate();
void allowUpdate();

int getLength();
Lane* seekLane(bool next);
void updateCarChangeLane(bool next);
void putCar(Car* car, int position);
bool isUsed(int position, int length);
bool getCrossInfo();
void setCrossInfo(bool set);
void changeLight();
void setTrafficLight(TrafficLight* trafficLight);
TrafficLight* getTrafficLight();
void setSpawnProbability(int probability);
void trySpawn();
int getVehicleLength();
```

World

```
static int length;
static int probability;
static int maxVelocity;
static int maxLength;

static int laneChangeProbabilityLow;
static int laneChangeProbabilityHigh;
```

```
static void initWorldVariables(int probability, int length, int
maxVelocity, int maxLength, int laneChangeProbabilityLow, int
LaneChangeProbabilityHigh);
```

Car

```
Lane* lane;
int length;
int position;
int velocity;
int id;
bool changedLane = false;
Lane* destination = nullptr;
bool doAccelerate = true;
bool alreadyUpdated = false;
```

```
static int count;
Car();
Car(Lane* road, int length);
~Car();
void update();
int checkFrontDistance();
int getLength();
Car* copy(Lane* lane);
void changeLane(Lane * lane);
void putInLane(Lane* lane);
Lane* doChangeLane(bool next);
void setDestination(Lane* destination);
Lane* getDestination();
int getFrontDistance();
void setLane(Lane* lane);
void setPosition(int position);
int moveSmooth();
void setUpdated(bool updated);
```