Imperial College
London

FINAL REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

LPS-visualisation

Author:
Jinwei Zhang (CID: 01540854)

Date: August 16, 2019

Contents

1 Introduction

LPS stand for Logic Production Systems, it is a "logic-based framework for programming databases and AI (intelligent agent) applications proposed by Robert Kowalski and Fariba Sadri"[1]. In their papers, they mentioned that LPS combines both production rules and logic program in a single logic-based framework. Production rules have a model-theoretic semantics, and logic programs manipulate a database of facts destructively[2].

JavaScript often abbreviated as JS, is a high-level, interpreted scripting language that conforms to the ECMAScript specification. HTML, CSS and JavaScript are three of the core technologies of the World Wide Web. JavaScript enables interactive web pages and is an essential part of web applications. JS is the top language based on repositories on Github as shown in Fig. 1

This material on JS belongs elsewhere - eg in the background or implementation, where you use it.

The introduction must set the scene for your project. To start by describing technologies is not a good idea.

After the first paragraph, say that LPS has been used to model ..... (put here whatever you like), and say that this project shows how LPS can model traffic behaviour norms. In particular, it has implemented models and behavioural norms including "standard traffic turning left or right", T-junctions, traffic-light controlled junctions, overtaking and traffic following, roundabouts, etc.

Then you can say LPS models can be simulated to show the behaviour captured in a model using a web-based javascript program and, for this application, animations have been written in PIXI.

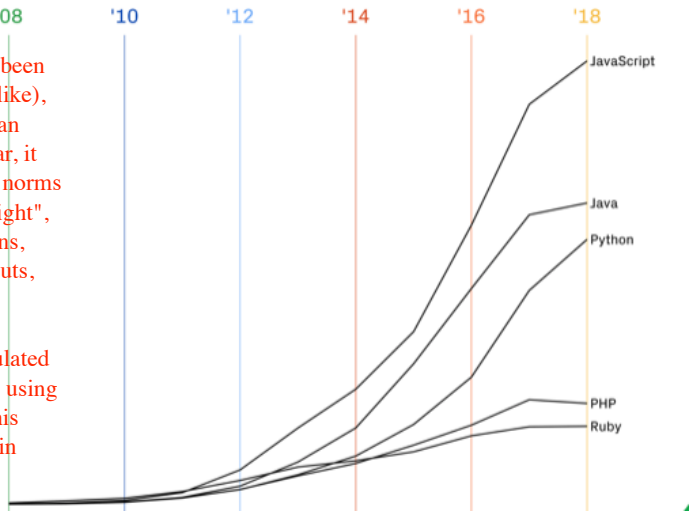


Figure 1: Top programming languages by repositories created, 2008-2018,JavaScript tops the list for the language with the most contributors in public and private repositories from year 2009.

Node.js was written initially by Ryan Dahl in 2009, is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser.

As LPS describes a framework, it has already been implemented in different programming languages. LPS has been extended onto SWISH – a SWI-Prolog web application – by Wielmaker et al.[6] Also LPS has been implemented in Javascript by Sam yong at a Msc project[]. There are various animation engine using javascript on the market such as PIXI which is the fastest, most flexible 2D WebGL renderer.

With the mature implemet of LPS in Javascript achieving function of animating LPS program become a greater further work for programmer to write LPS program which has large number of cycles.

1.1 Aim

The main objective of this project is to apply and extend a JavaScript implementation of a logic-based language LPS to a multi-agent simulation of self-driving cars. The same set of rules will be applied to all cars to achieve a certain goal such as reaching out to a goal destination, avoiding accidents. Additionally, cars should be complying with traffic regulations. To achieve this objective, the following were required:

These first three are not aims as such. The are aspects that are necessary for the aims which are the last three points.

- Understand the LPS syntax and semantics
- Understand the LPS.js interpreter where the back-end will be running
- Understand Node.js run-time where the back-end will be tested and running on.
- Design an easy-to-use and well documented front end and back end.
- Design a suitable system for the LPS visualization
- Preparing different cases of the different scenarios for traffic as described in section

This project partially relied on Msc student's project "Implementation of Logic Production System (LPS) in JavaScript" [3], specifically, LPS.js [3] which is the interpreter engine for LPS program. Based on the problem of the previous work such as undocumented code, bug in peer-to-peer function, a new system should be developed by abandoning most of the previous visualisation work. This project intends to record the state of the fluent run by LPS.js engine and re-implement the visualization in the front-end side.

Wrong place for above paragraph.
I believe we already suggested that you need to put all technology issues separate form the project aims and LPS, especially since you have not used much of the previous code. You need to have somewhere a very clear description of what technology is used and what was not used, and why.

1.2 Requirement

There were several requirements before the design and implementation of visualization of LPS.

1. Extensible
   The implementation should allow developer and third-parties to use or extend the features using the provided open window/API without having to modify the source code. For contributor the source code should be well documented such that developer can easy to follow and contribute.

2. User-friendly
   The implementation is user-oriented. This means that the program should be robust enough for non-experienced user to get involved easily by some basic tutorials. This also means the program should be less flexible in order to avoid unnecessary bug.

3. Efficiency
   Efficiency means the code should be efficient to development also means low coupled and low coherence. Program should be hosted on a server and running in a reasonable amount of time. asynchronous function are considered to be used in stead synchronous using promised feature in order to avoid unnecessary waiting time.

4. Testability
   Since the implemented program may have issue of latency, wrong data structure parsing format etc. The testing code should be formated as to test class types, program logic and functionality. Moreover, the stress testing should be performed to test the durability of the problem and analyse the problem.

1.3 Report structure

This report consists of 5 sections. Section 2 of this report Section 3 is the intermediate report which is the experiment that I have carried out so far. Section 4 is the criteria of this project and section 5 is about the plan in the future.
This report is structured in the following manner:

- **Section 2** is about the background of this project which consists of the detail about LPS, different traffic norms iand previous related work (LPS.js). Also some experiment carried out in the early stage.

- **Section 3** is the implementation of the project including the LPS program, front-end side of data processing, data formatting. Back-end side of server side programming.

- **Section 4** details the experimentation results of LPS visualisation, particularly in the latency in the goal achieving strategies, and evaluation of its performance and applications.

- **Section 5** concludes the report, and recommends some future extension and research rele- vant to this project and area of research

The whole system is named as LPS visualiser. The report has written with the assumption of the reader is familiar with predicate logic and first-order logic (FOL). As with Prolog's naming convention, predicate names used in this report and in LPS would start with a lower case letter while variable names start either with an underscore or an upper case letter.

2 Background

2.1 LPS

LPS stands for logic production system "it combines production rules and logic program in a single logic-based framework".Operational and model-theoretic semantics is associate to production rules and a database of facts extends the logic programs [2].
Production rules[2]:

- Production rules are typically ***condition*** $\rightarrow$ ***actions*** like the form of condition in logic.
- Production rules are consist of three different kinds where are reactive rules, forward chaining logic rules and goal-reduction rules.

Production system [2]:

Not sure bullet points are right format here. One would normally write in paragraphs.

- A typical production system combines facts which is a working memory of atomic sentences, and production rules in the form of ***condition*** $\rightarrow$ ***actions*** as we described above.
- The working memory can be viewed as a model-theoretic structure
- The language of the condition of a rule is generally a subset of first-order logic

production system framework[4]:

- Logic production system framework is in the form of ***<R, L, D>***
- **R** stands for Reactive Rules, it is in the form of $\forall X[$ antecedent $\rightarrow \exists Y$ consequent $]$
- **L** stands for logic programming. $\boldsymbol{L} = L_{int} \cup L_{\text{events}} \cup L_{\text{aux}}$. It is the states and events from the agents view. It is consists of 1.definition of intentional predicate. 2.composite event 3.auxiliary predicate (never change overtime).
- "Events include external events and an agent's own internally generated actions. Events are represented by atomic formulas including time and possibly other parameters".
- **D** stands for domain theory $\boldsymbol{D} = D_{pre} \cup D_{post}$. It is the precondition and the post condition of state transaction.

Operational Semantics[4]:

You need to give some illustrative short example of an LPS framework. As it stands, this doesn't clarify what LPS is for anyone who is unfamiliar with it.

- The operational semantics (OS) is a non-terminating cycle. the external events and the agent's successfully executed actions are merged together in OS. The state is destructively updated when the agent determined what to do next.
- Goal States Gi (set of goal trees) will be maintained in addition to the current state Si.

My suggestion would be to give one or more short examples that allow you to point out in the example the various features of LPS which you have defined in this section. For instance, what is a state Si, or a goal state? Quite often, it is easier to explain these things through examples.

2.2 LPS in JavaScript understandable format

There is no open API as

2.3 Traffic Regulation

Self-driving cars already navigate US highways and many other countries around the world. The safety become a big issue for self-driving cars. According to the research of "traffic management systems" 22% out of the students participate in the survey had been injured and 13% had been admitted into hospital for 9.3 days [5][6]. Hence designing a well functioning self-driving car system is the core of the project. Here are some basic scenarios and relevant researches that need to be addressed when making the rules for the self-driving cars to form a safe decision.

- Roundabouts

  Roundabouts are where number of junctions meets from different direction in a one-way circular road.
  Roundabouts are designed to keep the traffic flow in clockwise or counter-clockwise depends on the country
  The circular road is a street and maybe made up of one or more lanes.
  The traffic entering the roundabouts must give way to traffic already on the roundabouts approaching from the right

- Uncontrolled T-junction

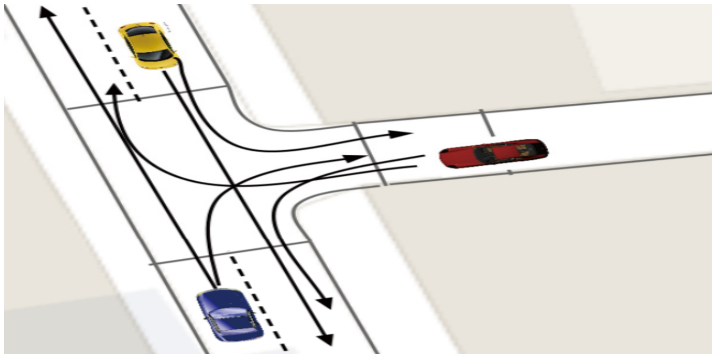


Figure 2: This is a uncontrolled T junction from paper "Ontology-Based Driving Decision Making"[7] , as showed in the picture the solid line represent the intention of each car. In this particular graph there are 6 different cases that might cause collision.

T-Junction is where three road meets from different direction in a T shape road.

T-Junction may be unmarked or marked, however, cars need to either 'Give way' (give priority to traffic on the main road) or 'Stop'(stop before the solid white line) based on the sign before joining the main road

If the road is uncontrolled with no sign, according to the simulation experiment in research"Ontology-Based Driving Decision Making" [7]. There are six possible paths overall and two maximum upcoming collision for the blue car as shown in fig. 2. The vehicle need to follow the SWRL Right-Of-Way rules [7] which are used for inferring rules to make safe driving decisions. We only consider the nearest vehicles on each lane or intersection and the other following vehicles are neglected. At a different timestamp,the neglected cars will be re-considered when they become the nearest to the blue vehicle.

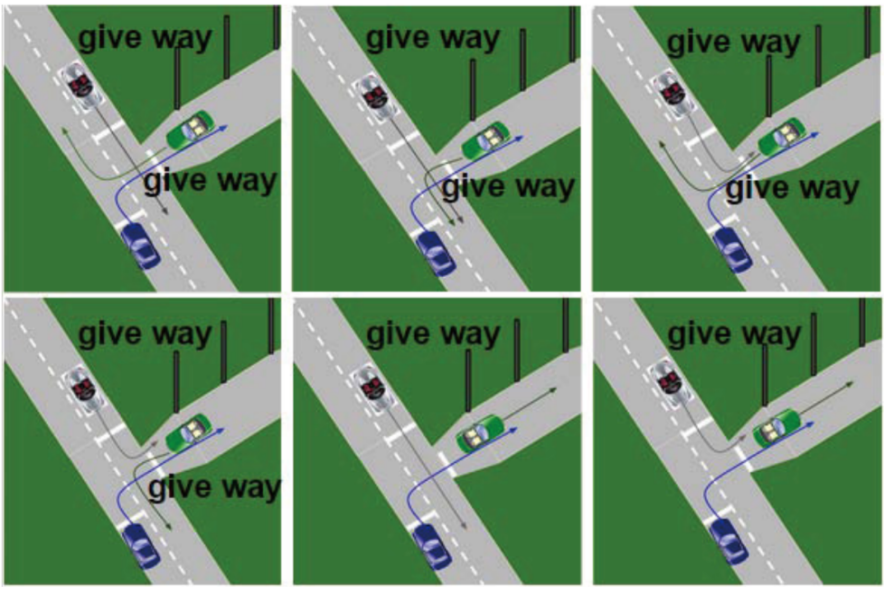


Figure 3: shows a detailed version of 6 different decision making in uncontrolled T-junction from the paper "Ontology-Based Driving Decision Making"[7]. In first four cases, it shows the blue vehicle should give way to both cars because one vehicle is driving straight and the other vehicle is driving out from narrow road to the wider road.

In general, certain rule should be applied to this particular traffic situation so that vehicles are able to make the right decision accordingly.
A car driving straight has a higher priority than a car turning right or left at an uncontrolled intersection.
If a car is going to drive into a narrow two-way road, it should wait for the other car which is going to drive out from the narrow two-way road.

- cross-junction

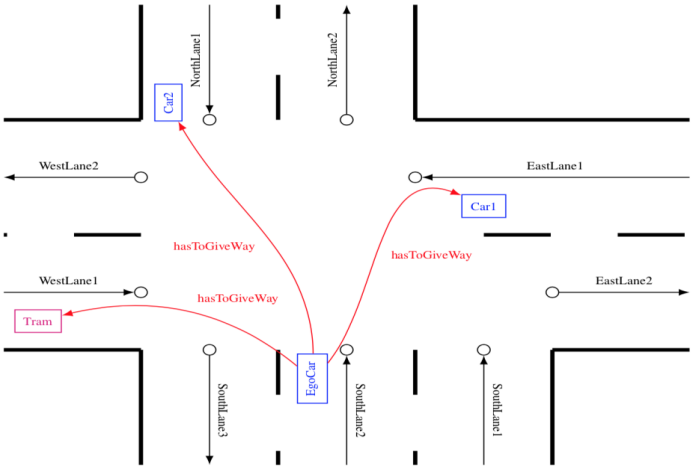


Figure 4: Cross-junction is the place where two road cross with a perpendicular angles as shown in Fig. 4. Cars must give way to traffic on the main road when emerging from a junction with broken white lines across the road.

In the research of "Traffic Scene Modeling" [8]. A sample set of rules for uncontrolled cross junction in SWRL can be written as below. Similar rules can also be formalised in LPS in the visulization project.

Yuo need to say in English what this expresses.
For instance, why would only vehicle 1 be a part of the scene?

$$
\begin{aligned}
&UncontrolledIntersection(?scen1) \\
&\wedge Vehicle(?vehicle1) \wedge Vehicle(?vehicle2) \\
&\wedge isPartOf(?vehicle1, ?scen1) \\
&\wedge hasConflictingConnector(?c1, ?c2) \\
&\wedge hasIntention(?vehicle1, ?c1) \wedge hasIntention(?vehicle2, ?c2) \\
&\Rightarrow hasToGiveWay(?vehicle1, ?vehicle2)
\end{aligned}
$$

Figure 5: cross junction rules
[8]

- Junctions controlled by traffic lights
  Vehicle must stop behind the ‘Stop’ line unless the light is green.
  Vehicle must not move forward over the stop line when the red light is showing [9]. Traffic light can also be treated like an agent which will also have a set of rules and being able to communicate with the vehicle.

- Overtaking

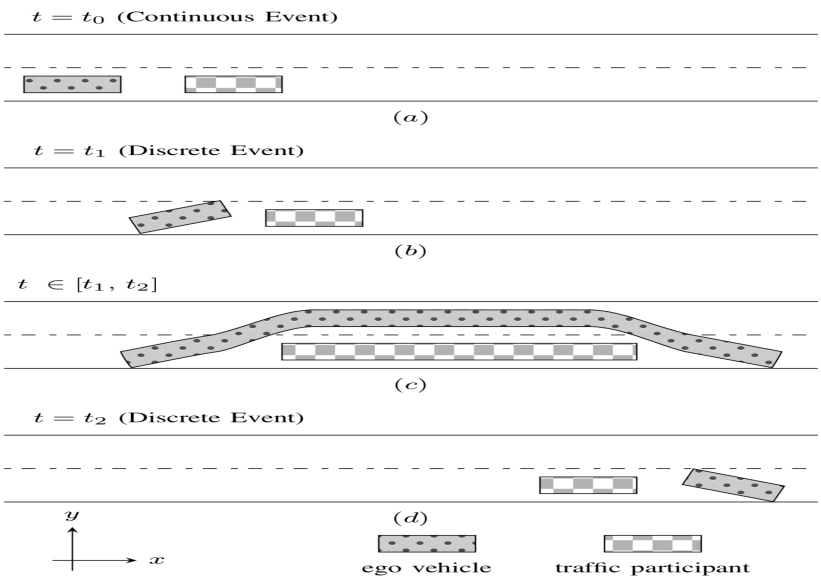


Figure 6: overtaking case
[10]

Fig. 6 is an illustration of overtaking scenario. When we overtake another traffic participant, the other lane must be free of any traffic occupancy. The ego vehicle in Fig. 6 have to leave a sufficient distance before returning to the original lane. In each time cycle, the position of the two cars should not overlap [10].

According to the research, we need to consider about the selection of most appropriate lane and the selection of velocity in a two lane overtaking case [11]. The mix of logic and integer linear inequalities should also be considered. One example is shown as fig. 10 below.

The vehicle can not overtaking if the road has any restriction such as 'No Overtaking' sign etc.

You should only explain here what technology you actually use.That is why a technology chapter is handy as you can say in that why you use what you do, and why you don't use any of the other possible technologies.

2.4 technology

2.4.1 swish visualisation

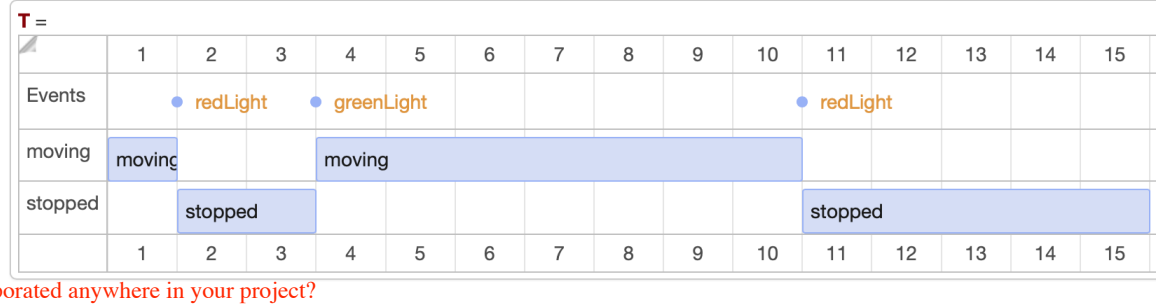
Different visualisation has been experienced at this stage. This section is the demonstration of using SWISH which is implemented in SWI Prolog[12] and LPS.studio to visualise the lps program. The example visualization is on SWISH .

```
maxTime(15).
% set of facts
fluents moving, stopped.
% cause state transition via init fact terminate fact
events redLight,greenLight.
% describes the facts that are true at time 1.
initially moving.

% describe events taking place in the transition from one time point to the next.
observe redLight from 1 to 2.
observe greenLight from 3 to 4.
observe redLight from 10 to 11.

% describe the laws of cause and effect
redLight initiates  stopped if moving.
redLight terminates moving if moving.
greenLight initiates moving if stopped.
greenLight terminates stopped if stopped.
```

```
?- go(T)
```



Is this incorporated anywhere in your project? If not, it could still be helpful in showing the traffic behaviours

Figure 7: swish visualisation.

Fig. 7 shows the swish visualisation of the fluent in different time cycle. The brief idea is to show the change of the fluent on the timeline. Initially the moving fluent is activated. From time 1 to time 2 the red light event is observed so the stopped state will be initiated and moving state will be terminated. Then the green light is on from time T3 to T4 so the moving state will be active again and the stopped state will be terminate. In the end, from time 10 to time 11 the red light event will again be observed so the state will again switch back to stopped from moving.

2.4.2 LPS.js

LPS.js is the JavaScript interpreter for LPS. Most of the function works fine except the peer-to -peer.

2.4.3 LPS studio

LPS studio is the visualisation tool for LPS program. Since the p2p function is not working at this moment. P2P services is considered to be re-implemented in the LPS visualisation system. LPS studio has some problems that I have encountered, including the conflict between the print statement and animation event. The animation malfunctioning in the recursion call etc.

1. One car moving on straight line scenario

Figure 8: One car moving case.

You need to introduce this language!

Fig. 8 shows a car that can move along a straight line, based on the first SWISH traffic light example. Initially, the car is given a moving state. From time T1 to T2 the traffic light is red so the car is at stopped state. The state of the car will be changing the same as the previous example. The reactive rules state whenever the car is at moving state the car will update its location and move straightly right by 64 pixels.

```
% set of facts.
fluents([moving, stopped, waitting, loc(Car,X,Y)]).
% cause state transition via init fact terminate fact
events ([redLight,greenLight]).
% action to change fluents
actions([arrival,carMoveTo(Car,NewX, B)]).

% describes the facts that are true at time 1
initially([moving, loc(car,150,340)]).

% describe events taking place in the transition from one time
   point to the next
observe(redLight,1,2).
observe(greenLight,3,4).
observe(redLight,7,8).
observe(greenLight,9,10).

%reactive rule
moving(T1),loc(car,A,B) ->
  carMoveTo(car, A + 100, B) from T1 to T2.

updates(carMoveTo(CAR, X, Y), loc(CAR,_, Y), loc(CAR,X, Y)).
initiates(redLight, waitting).
terminates(redLight, moving).
initiates(greenLight, moving).
```

```
25 terminates(greenLight, waitting).
26 initiates(arrival, stopped).
27 terminates(arrival, moving).
```

Adding the code below and change the reactive rule will make the animation work. Firstly define the car object, then adding the lpsAnimateMoveObject micro-event to make the car move on the canvas by increase 100 pixels once per time cycle.

```
1  lpsDefineObject(car, image, [
2    position(150, 340),
3    size(64, 64),
4    isHidden(0),
5    image(car),
6    zIndex(100)
7  ]).
8  moving(T1),loc(car,A,B) ->
9    carMoveTo(car, A + 100, B) from T1 to T2,
10   lpsAnimateMoveObject(car, 2, A + 100, B) from T2 to T3.
```

2. T-junction scenario

This is definitely not background. It is your project!
Background means existing theory or frameworks which are necessary to know in order to understand the project. It does not include any work you have done as part of the project.

Two car moving towards the goal location in a T-junction case has been worked out as fig. 10. After experienced this scenario couples of problem has been found: If the lpsAnimateMoveObject be treated as the condition for update. The program will fail. If the lpsAnimateMoveObject was placed in a recursive call the animation will fail however the lps program will not fail. More reasons please refer to the repository: LPS studio[13]: `https://github.com/lps-js/lps-studio`

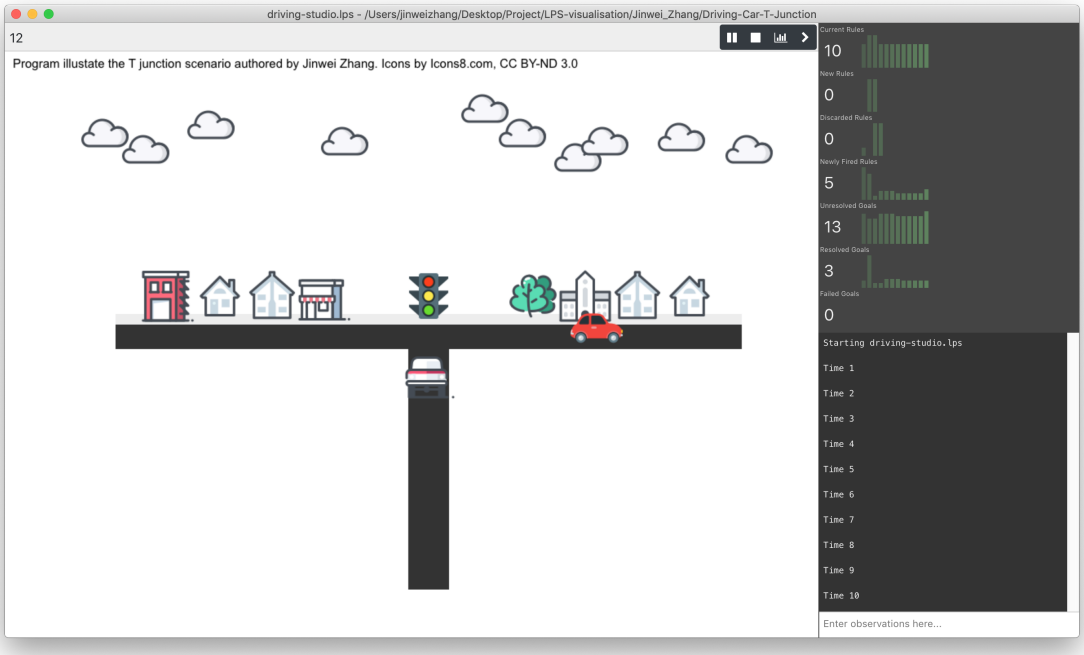


Figure 9: Two cars moving in a T junction.

Fig. 10 is the T-junction scenario it shows the red car is trying to move 64 pixels right each cycle along the horizontal road. The other car is trying to move up and turn left. Initially, two cars are all given a start position and a moving state. The reactive rule is keeping checking the state of the two cars and the distance between each other. If the car has not reach to the goal and the distance is valid, the reactive rule will fire. The other car give way to the red car if the Manhattan distance is smaller than 200 and turn left if the left is clear and the Manhattan distance is greater than 200. Mention that the traffic light is red from time T4 to T5 when the car is at stopped state and turns back to green from time T9 to T10 when then car is back to moving state.

```
% set of facts.
fluents([moving(Obj), stop(Obj), waitting(Obj),car_location(Obj,
    _,_)]).
% cause state transition via init fact terminate fact
events ([redLight, greenLight]).
actions ([carMoveTo(Obj,Loc_X, Loc_Y)]).
% describes the facts that are true at time 1
initially([moving(car1),moving(car2),car_location(car1, 150,
    340),car_location(car2, 520, 650)]).

% describe events taking place in the transition from one time
    point to the next
% event redLight will happened from T4 to T5
observe(redLight,4,5).
% event greenLight will happened from T8 to T10
% all the car stop moving between time 5 and time 9
```

```
observe(greenLight,9,10).
%goal location of the two cars
goal_location(car1, 800, 340).
goal_location(car2, 150, 340).

%the reactive rules for car1, checking if car1 has reached it's goal

goal_location(car1, A, B), car_location(car1,X,Y,T), X@<A, moving(car1,T) ->
   NextX = X + 64,
   carMoveTo(car1, NextX,Y) from T to _.

%the reactive rules for car2, checking the distance between car1 and car2 and whether it need to move up
goal_location(car2, A, B),
  car_location(car2,X,Y,T), car_location(car1,P,Q,T),abs(X - P)+ abs(Y - Q) > 200, Y>=B, moving(car2,T) ->
   NextY = Y - 64,
   carMoveTo(car2, X,NextY) from T to _.
%the reactive rules for car2, checking the distance between car1 and car2 and whether it can move to the left
goal_location(car2, A, B),
 car_location(car2,X,Y,T), car_location(car1,P,Q,T),abs(X - P)+ abs(Y - Q) > 200, X@>A, Y<=B, moving(car2,T) ->
  NextX = X - 64,
  carMoveTo(car2, NextX,Y) from T to _.

% carMoveTo action will terminates the old car_location and initiates a new car_location to X Y
updates(carMoveTo(Obj,X, Y), car_location(Obj,_, _), car_location(Obj,X, Y)).

initiates(redLight, waitting(Obj)).
terminates(redLight, moving(Obj)) .
initiates(greenLight, moving(Obj)).
terminates(greenLight, waitting(Obj)) .
```

Listing 1: prolog

Adding the code below and change the reactive rule will make the animation work.Firstly define the car1 object and car2 object, then adding the lpsAnimateMoveObject macro-event to make the car move. The example of making car1 move is shown below.

```
lpsDefineObject(car1, image, [position(150, 340),size(64, 64), isHidden(0),image(car1),zIndex(100)]).
lpsDefineObject(car2, image, [position(520, 650),size(64, 64), isHidden(0),image(car2),zIndex(100)]).

%car moving sepraate the first car and the second car
goal_location(car1, A, B), car_location(car1,X,Y,T), X@<A, moving(car1,T) ->
   NextX = X + 64,
   carMoveTo(car1, NextX,Y) from T to _,
```

```
8    lpsAnimateMoveObject(car1, 2, NextX, Y) from T to _.
```

2.4.4 LPS Cli

LPS CLI tool gives the interface of testing and running the LPS interpreter in command line level . It is a lower level of runtime implement of lps.js. However, the code is not documented, so understanding the logic in the code is time-consuming.

2.4.5 Javascript asynchronous programming

JavaScript is a single-threaded programming language which means only one thing can happen at a time [14]. However, running the LPS animation might take some time to process while blocking the main thread making the web page unresponsive. That's where asynchronous JavaScript comes into play. Using asynchronous JavaScript (such as callbacks, promises, and async/await), users can perform other function such as stop the program, clear the program, delete the content while the LPS program is running. The asynchronous work as follow:

Execution Context An Execution Context is an abstract concept that holds information about the environment within which the current code is being executed [15]. Whenever any code is run in JavaScript, it's run inside an execution context. The global execution context is created before any code is executed.Whenever a function is executed (or called/invoked, these are all synonyms), a new execution context gets created. Every execution context provides this keyword, which points to an object to which the current code that's being executed belongs.

asynchronous callbacks. JavaScript has a call stack which is a stack with a LIFO (Last in, First out) structure. The call stack is used to store all the execution context created during the code execution. Since JavaScript is a single-threaded programming language it only have a single call stack. When it comes to asynchronous execution asynchronous callbacks is introduced to understand callback event loop and the callback queue(message queue/task queue) need to addressed.

event loop The job of the Event loop is to look into the call stack and determine if the call stack is empty or not. If the call stack is empty, it looks into the message queue to see if there's any pending callback waiting to be executed. Given the example below as an example.

```
1 const networkRequest = () => {
2   setTimeout(() => {
3     console.log('Async Code');
4   }, 2000);
5 };
6 console.log('Hello World');
7 networkRequest();
8 console.log('The End');
```

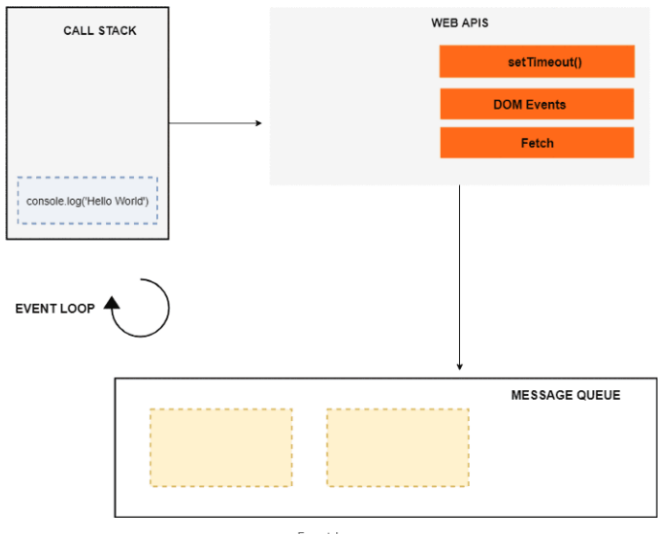


Figure 10: The execution of the code: firstly browser loads console.log('Hello World') is pushed to the stack and popped off the stack after it's finished. Next, a call to networkRequest() is encountered, so it's pushed to the top of the stack. Next setTimeout() function is called, so it's pushed to the top of the stack. The setTimeout() has two arguments: 1) callback and 2) time in milliseconds (ms). The setTimeout() method starts a timer of 2s in the web APIs environment. At this point, the setTimeout() has finished and it's popped off from the stack. After it, console.log('The End') is pushed to the stack, executed and removed from the stack after its completion. Meanwhile, the timer has expired, now the callback is pushed to the message queue. But the callback is not immediately executed, and that's where the event loop kicks in.

The Event Loop The job of the Event loop is to look into the call stack and determine if the call stack is empty or not. If the call stack is empty, it looks into the message queue to see if there's any pending callback waiting to be executed.

I am completely lost I'm afraid, and do not know which parts are new, which already existed, whether you used them or not, etc. Please make it clearer.

DOM Events DOM (Document Object Model) events are an actions that occurs as a result of the user action or as result of state change of the elements of a DOM tree. Examples of HTML DOM events such as: user clicks the mouse on a button, When a web page has loaded. In case of DOM events, the event listener sits in the web APIs environment waiting for a certain event (click event in this case) to happen, and when that event happens, then the callback function is placed in the message queue waiting to be executed.

Promise callbacks being used repeatedly and irresponsibly by the programmer can be lead to a phenomenon called "pyramid of doom" or "callback hell"[16]. The callback hell code can become hard to maintain and flooded scope with irrelevant variables, making it difficult to debug and maintain.ES6 introduced the concept of job queue/micro-task queue which is used by Promises in JavaScript. The difference between the message queue and the job queue is that the job queue has a higher priority than the message queue, which means that promise jobs inside the job queue/

micro-task queue will be executed before the callbacks inside the message queue. In conclusion: the event loop will first empty the micro-task queue before executing any callback in the message queue.

2.4.6 Node.js

As an asynchronous event driven JavaScript run-time, Node is designed to build salable network applications. Node presents an event loop as a runtime construct as we describe in section "The Event Loop". In other systems there is always a blocking call to start the event-loop. Typically behavior is defined through callbacks at the beginning of a script and at the end starts a server through a blocking call like EventMachine::run(). In Node there is no such start-the-event-loop call. Node simply enters the event loop after executing the input script. Node exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript — the event loop is hidden from the user. HTTP is a first class citizen in Node, designed with streaming and low latency in mind. This makes Node well suited for the foundation of a web library or framework.

2.4.7 PIXI.js

PixiJS is a rendering library that will allow user to create rich, interactive graphics, cross platform applications, and games without having to dive into the WebGL API or deal with browser and device compatibility.
PixiJS has full WebGL support and seamlessly falls back to HTML5's canvas if needed. As a framework, PixiJS is a tool for authoring interactive content, especially with the move away from Adobe Flash in recent years. Since pixi.js for animate graphics rich, interactive websites, applications it can be suitable for implementing a animation for LPS program. Moreover, it is also cross-platform compatibility and easy to hook with the LPS interpreter lps.js.

2.4.8 Express.js and REST API

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. With a myriad of HTTP utility methods and middle-ware at your disposal, creating a robust API is quick and easy. It provided a range of function for user to develop and build the back-end side of the web application.
A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data.
A RESTful API – also referred to as a RESTful web service – is based on representational state transfer (REST) technology, an architectural style and approach to communications often used in web services development.
REST technology is generally preferred to the more robust Simple Object Access Protocol (SOAP) technology because REST leverages less bandwidth, making it more suitable for internet usage.

2.4.9 MongoDB

MongoDB is a document database, which means it stores data in JSON-like documents. Since a full functional web application with rest API is going to be build. Database is essential for store not only the user information but also the LPS program potentially. MongoDB is one the most light weight database and it support cloud storage.

3 Implementation

3.1 Overview

This part is about the implementation of the LPS visualisation. This section consist of 5 part. The second part is about the structure of the project including the link between the front end and back-end side. The third is about the LPS program implement to fit various traffic situation. The Fourth part is about the front side LPS program parsing, data structure processing and animation. The fifth part will be the server side programming including using technology Express and some user authentication tools. Since the main purpose of this project is to visualise the LPS program in traffic scenario lively. There are several issue that need to be addressed before designing and implementing the program.

Rephrase - you have addressed these, presumably!

1. How will the LPS program be prased into JavaScript since there is no interface of write LPS code direct in JavaScript format.

   To adding API into LPS interpreter is very time consuming because it needs to understand and modify the LPS.js directly and the interpreter (lps.js) does not encapsulate the key function in (Engine.js) also address by Sam Yong “lps.js does not support the construction of LPS programs by JS without explicitly writing LPS code in order to reduce the amount of code base and API implementation needed.” The idea of formatting LPS directly in JavaScript was abandoned very early. To solve the data parse issue we parsed data direct from the LPS code (file with extension .lps) by file proceeding in a asynchronous way.

   Does this mean you wrote your own parser? If so, I didn't realise it. If not, you just need to say what you did use.

2. What front-end tool is going to be used to achieve live animation. There is a animation tool which has been implemented in Node.js Electron called LPS studio however there are variety miserable bugs such as (placing print before action predication causing crash of the program, unable to work with recursion etc). The idea of using LPS studio is abandoned. After week of research, PIXI.js was found very suitable to animate the LPS program. This lead to another problem wehter to use pixi direct in HTML or use it in node.js by create another canvas. By some experiment, using it directly in HTML was more light weighted which was PIXI.js the front-end side animation side original design to.

   Don't phrase these as questions. Give an overview diagram with blocks for the various programs and links for the data flows. You can then describe each block in order explaining why you chose the method of implementation that you did.

3. What information of LPS program need to be parsed into the program. These three question were all tucked down during a long time research. There is one open interface that is available which is the

I'm afraid the above is rather opaque.

```
1 const LPS = require('lps');
2 LPS.loadString('...')
3   .then((engine) => {
4       engine.run();
5   });
```

   The data that we are parse onto the web is then solved. The only issue we need to solve is to formatting different fluents into different data structure and

capture the change of fluent in each running cycle, detail will be illustrated in the following subsection.

3.2 Structure design

This should be before the above section. Where is the parser? Maybe there isn't one? I really didn't follow the paragraph on parsing above.

The high-level system diagram of LPS visualiser is shown as follow:

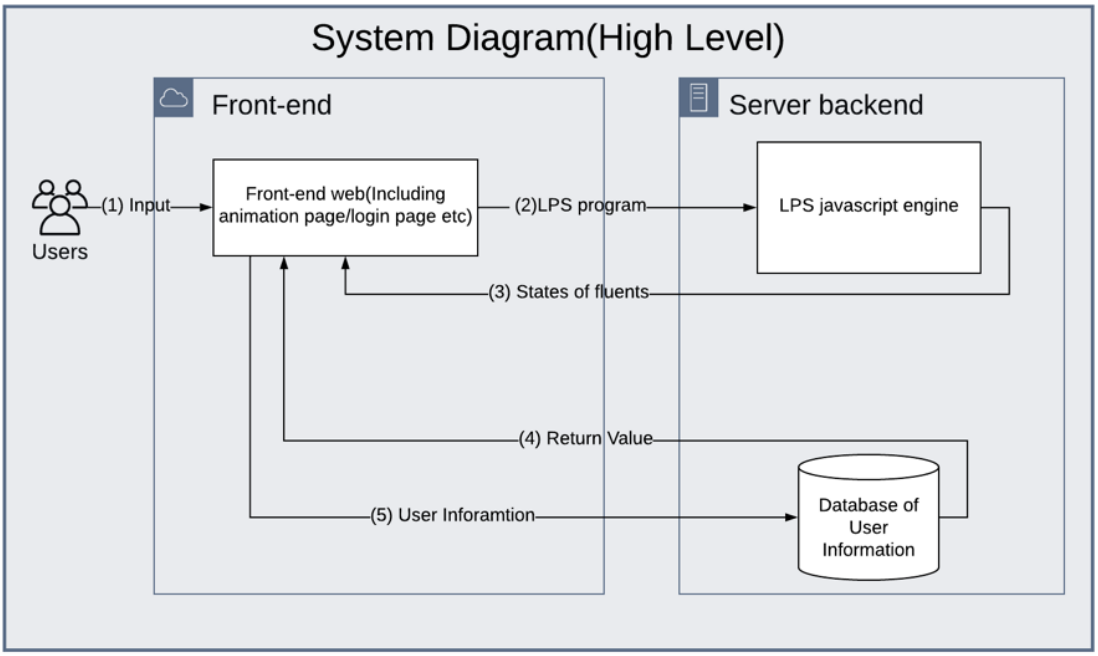


Figure 11: shows the system diagram with the event flow and data flow. (1) Means any input to the system such as typed in user information, loaded LPS code, mouse dragging event, typing LPS program in text area and so on. After the input event happened the corresponding page need to handle these responses or input accordingly. that the LPS program should be in JavaScript syntax. (2) The main job of the LPS visualiser is to pass the LPS program into interpreter (The LPS.js) running as a JavaScript library. After that the changing state of fluent(as No.3 shown) will be passing onto the web page while the LPS program is running benefited to the JavaScript concurrency model. (4) The interaction of the back-end program and the also include the login and register part. User need to either register as a new user or login. After this part user information in the database will be retrieved (as No.5 shown), and REST API will comes to play. REST API will return to the main visualization page if user information is valid or stay in the login page if user information is not valid.

Yes, this is more informative.

3.3 LPS program

3.3.1 Overview

The LPS program is the input program. For the traffic the LPS program should not only follow the structure of the interpreter understandable way as we discussed in section (?) also it should be easy to be parsed into the PIXI animation engine. Hence

the format is defined as below (User can replace any Upper-case text to real instance number or predicate:

```
% we assume the destination is reachable
maxTime(Number).
cycleInterval(Number).

fluents([
   Set of fluent predicate defined
]).

events ([
  Set of event predicate
]).

Observe([
Set of observation predicate (happeded at a particular time)
]).

actions ([
  Set of actions predicate defined
]).

initially([
     Initial set of fluent predicate (true from cycle 1)
]).


Fact predicate (true forever)

Reactive Rules

Predicate Rules

Action (change fluent)
```

You need to explain about the canvas before talking about streets, coordinates etc. Give an example layout and explain how it would be represented. You can then explain that there are various junction types each one being explained in detail in the following subsections. Make clear how a layout with more than one junction would be encoded.

1. Define street(fact) as: street(StreetName, coordinate(X, Y), Width, Height, Number_of_lane) ie street(mainStreet , coordinate(100, 200), 900, 50, 1).Meaning there is a street called mainStreet with the top left corner (100,200). Width 900 height 50 and the number of lane is 1.

2. Define location of vehicle(fluent) as: location(Name_of_car, coordinate(X, Y), Direction) ie: location(car0, coordinate(150, 225), eastward) meaning car0 is located at (150,225) and facing eastward.

3. Define traffic light(fluent) as: trafficLight(coordinate(X, Y), Working_status, Color, FacingDirection) ie: trafficLight(coordinate(430, 475), red, westward, mainStreet) meaning there is a traffic light located at coordinate(430, 475) and it is showing red color at the moment, the traffic is facing

4. Define the goal of the Vehicle(fluent) as: goal(VehicleName, coordinate(X, Y)) ie: goal(car0,coordinate(980, 475)), meaning the goal of car0 is at (980 475)

5. Each vehicle can have multiple state(fluent) eg stopped(VehicleName) or moving(VehicleName) ie: moving(car0) meaning car0 is at state of moving.

6. Define the junction(fact) as: eg junction(JunctionName,coordinate(A, B),coordinate(C, D),coordinate(E, F),coordinate(G, H)). where (A, B), (C, D), (E, F), (G, H) are 4 different corners. Ie junction(tJuntion1,coordinate(450, 450),coordinate(500, 450),coordinate(450, 500),coordinate(500, 500)). There is a junction called tJuntion1 the four corner are (450, 450), (500, 450), (450, 500), (500, 500).

3.3.2 car moving straight

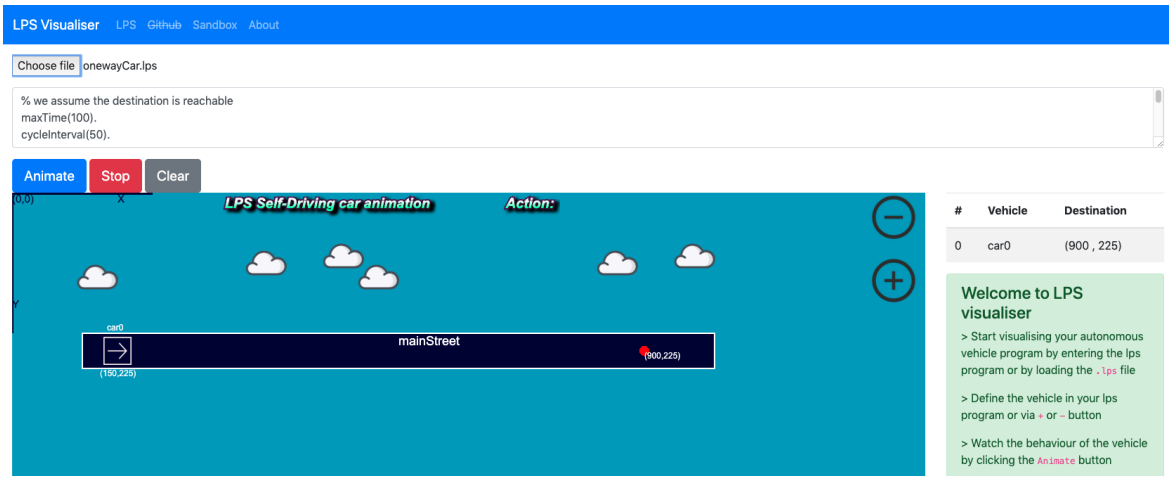


Figure 12: show a car can move straight towards the goal (red point)

```
% we assume the destination is reachable
maxTime(100).
cycleInterval(50).

loadModule('../scripts/module.js').

fluents([
    stopped(VehicleName),
    moving(VehicleName),
    coordinate(X, Y),
    location(VehicleName, coordinate(X, Y), Direction),
    trafficLight(coordinate(X, Y), Working_status, Color,
        FacingDirection),
    street(StreetName, coordinate(X, Y), Width, Height, Number_of_lane
        ),
    goal(VehicleName,coordinate(X, Y))
]).

actions ([
  step(Vehicle, NextPlace),
  turn(Vehicle, NewHeading),
  arrive(Vehicle)
```

```
21 ]).
22
23 initially([
24   moving(car0),
25   location(car0, coordinate(150, 225), eastward),
26   goal(car0,coordinate(900, 225))
27 ]).
28
29 street(mainStreet , coordinate(100, 200), 900, 50, 1).
30
31 cloud(coordinate(140,100)).
32 cloud(coordinate(250,90)).
33 cloud(coordinate(300,120)).
34 cloud(coordinate(640,100)).
35 cloud(coordinate(750,90)).
36 cloud(coordinate(800,120)).
37
38 goal(Vehicle,coordinate(A, B)) from _ to T,
39 location(Vehicle, coordinate(X, Y), Direction), A==X, B==Y,moving(
       Vehicle) at T ->
40   % testPrint(Vehicle+ ' we have arrived'),
41   arrive(Vehicle) at T.
42
43 goal(Vehicle,coordinate(A, B)) from _ to T,
44 location(Vehicle, coordinate(X, Y), Direction), A!=X at T ->
45   % need to find the right direction here
46   % driving forward
47     drive(Vehicle) from T to _.
48
49 goal(Vehicle,coordinate(A, B)) from _ to T,
50 location(Vehicle, coordinate(X, Y), Direction), B!=Y at T ->
51   % need to find the right direction here
52   % driving forward
53     drive(Vehicle) from T to _.
54
55 drive(Vehicle) from T to T1 <-
56   location(Vehicle, coordinate(X, Y), Direction),
57   Direction == northward,
58   NewY = Y - 10,
59   NextPlace = coordinate(X, NewY),
60   step(Vehicle, NextPlace) from T1 to T2.
61
62 drive(Vehicle) from T to T1 <-
63   location(Vehicle, coordinate(X, Y), Direction),
64   Direction == southward,
65   NewY = Y + 10,
66   NextPlace = coordinate(X, NewY),
67   step(Vehicle, NextPlace) from T1 to T2.
68
69 drive(Vehicle) from T to T1 <-
70   location(Vehicle, coordinate(X, Y), Direction),
71   Direction == westward,
72   NewX = X-10,
73   NextPlace = coordinate(NewX, Y),
74   step(Vehicle, NextPlace) from T1 to T2.
```

I think to have 2 pages of code without any description is not good practice. Once you have described about layouts and declarations, in previous subsection, you can focus here on the production rules. Presumably a lot of these clauses are common to many types of junction, in which case describe them first and just once. Then just put the junction specific parts here and explain each rule as you go.


```
75
76 drive(Vehicle) from T to T1 <-
77   location(Vehicle, coordinate(X, Y), Direction),
78   Direction == eastward,
79   NewX = X + 10,
80   NextPlace = coordinate(NewX, Y),
81   step(Vehicle, NextPlace) from T1 to T2.
82
83 % on(coordinate(X,Y),Street) <-
84
85
86
87 updates(step(Vehicle, NextPlace), location(Vehicle, OldPlace,
       Direction), location(Vehicle, NextPlace, Direction)).
88
89 updates(turn(Vehicle, NewHeading) , location(Vehicle, Place,
       OldHeading), location(Vehicle, Place, NewHeading)).
90
91 terminates(arrive(Vehicle), moving(Vehicle)).
92 initiates(arrive(Vehicle), stopped(Vehicle)).
93 % you  can not move (x++) if you are not facing eastward
```

The reactive rule checks if the goal has been reached if not then keep driving towards the current direction. When step forward step action is performed which will terminate the old location and initialize a new location. Once the goal was reached, the moving state will be terminated.

3.3.3 T junction with both narrow lane

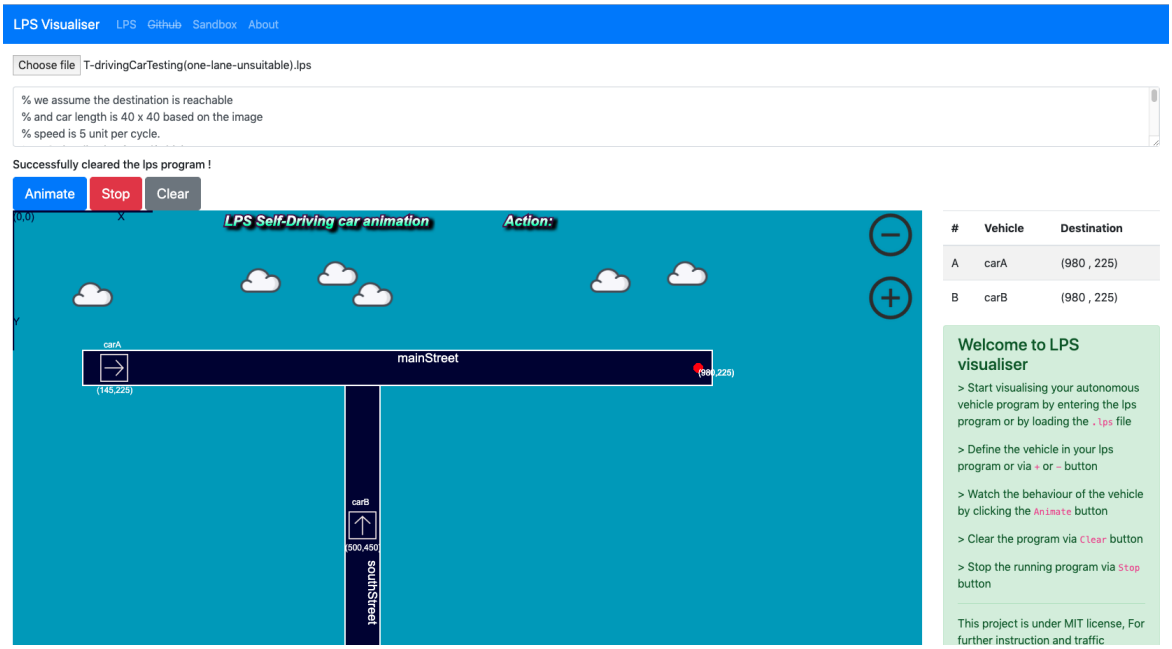


Figure 13: In the graph the red dot represent the goal location of the car. In this particular example both carA and carB has the same goal of (980 225). However when both car reached the T junction carB should give way to the carA because the car on the main road has higher priority in this case.

```
1              goal(Vehicle,coordinate(A, B)) from _ to T,
2          location(Vehicle, coordinate(X, Y), Direction),
3            moving(Vehicle) at T ->
4              % need to find the right direction here
5              direction(Vehicle) at T.
```

This reactive rule was added in this scenario because the vehicle need to check the direction

3.3.4 T junction with one narrow lane and a double lane

1. Two car:

Figure 14: This is similar to the previous T junction with both narrow lane however a new road and junction need to be defined in this case.

2. Three cars:

Figure 15: The three car in a T junction scenario is more completed. When cars meet at the T junction it need to make decision depends on the intention of the other car. However some rules are applied here: Assumption: carA intend to turn right or go straight. carB intend to turn left or right, carC intend to turn left or go straight

Based on the assumption, few rules are made:

(a) A car driving straight has a higher priority than a car turning right or left at an un-controlled T-junction.

(b) If a car is driving into a narrow road it should wait until other car drive out of the narrow.

(c) One car is turning left and another car is turning right to the same road at the same time, left turn has higher priority

```
% % There are 8 cases in total that your route will be blocked
% if Vehicle is at same horizontal level as its goal and there
    is another car in between
blockedRoute(Vehicle,Vehicle2) at T<-
  location(Vehicle, coordinate(X1, Y1), Direction),
  Direction == eastward,
  goal(Vehicle,coordinate(A1, B1)),
  location(Vehicle2, coordinate(X2, Y2), _),
  Vehicle != Vehicle2,
  Y1 == B1,
  Y2 == B1,
  X2 >= X1,
  X2 <= A1.
```

as described above, one example of vehicle2 blocked the route of vehicle code of is written as above This example describe about Vehicle is at same horizontal level as its goal and there is another car in between. However, there are 8 cases in the blocking road. There are 4 cases in car driving straight toward (toward north south east west) to the destination and there is a blocking car. there are

4 more cases about car in a L shape road these cases are car on the main road and turning left car on the main road is turning right. Car on the narrow road and turning left, car on the narrow road and turning right. As I illustrated in the graph Fig. 16:

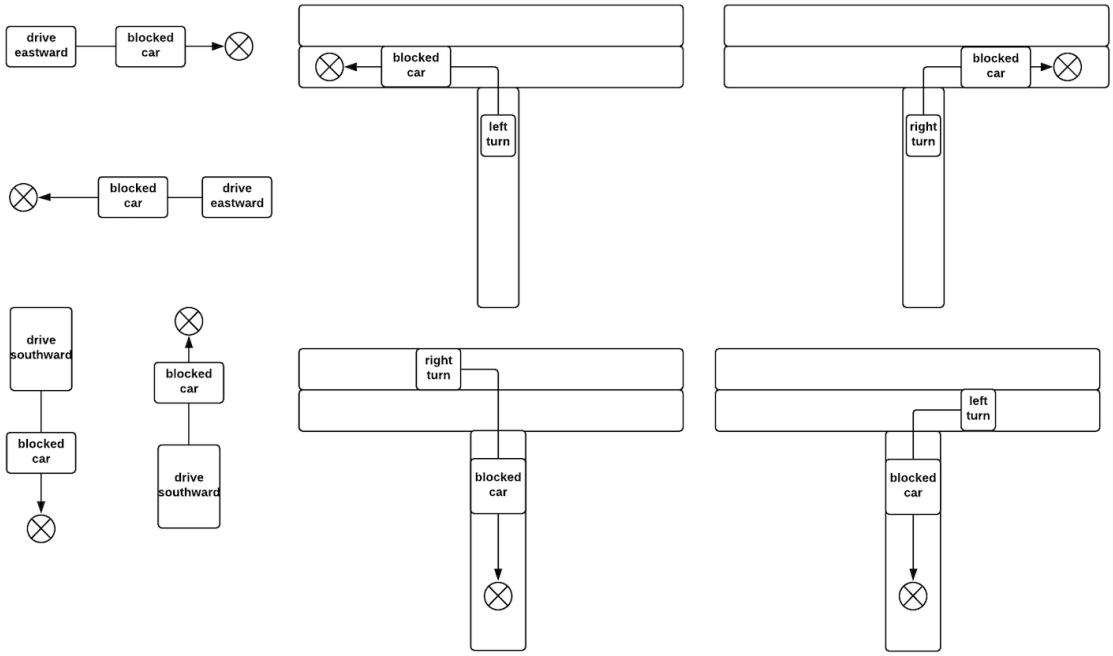


Figure 16

```
1
2 intention(VehicleName, Plan) <-
3   location(VehicleName, coordinate(X, Y), Direction),
4   goal(VehicleName,coordinate(A, B)),
5   Direction == northward,
6   Y > B,
7   X > A,
8   Plan = turnLeft.
```

Intention of the vehicle will also be added since vehicle meet at this particular junction need to make decision depends on the intention of other vehicles. The predicate of intention is also defined as follow:

```
1 % both car have clear routes but car not on main street can not
    step
2 <- step(Vehicle, NextPlace),
3   not onMainRoad(Vehicle) at T,
4   collisionPossible(Vehicle, Vehicle2) at T,
5   Vehicle != Vehicle2,
6   clearRoute(Vehicle) at T,
7   clearRoute(Vehicle2) at T.
```

Also the restriction is added in this case because the vehicle need to check wether the route is clear is not is cannot step ahead

This is rather better format and description etc.

```
% both car have clear routes but car on the narrow road can not
    step
% (car on main street has higher priority)
<- step(Vehicle, NextPlace),
  not onMainRoad(Vehicle) at T,
  collisionPossible(Vehicle, Vehicle2) at T,
  Vehicle != Vehicle2,
  onMainRoad(Vehicle2) at T,
  not blockedRoute(Vehicle,Vehicle2) at T,
  not blockedRoute(Vehicle2,Vehicle) at T.
```

The restriction of step action is will also be applied one of the example of both car have clear routes but car on the narrow road can not step is shown as above.

3.3.5 T junction with two double lanes

Figure 17: In this case the traffic light is introduced here which means car is not going making decision when there is more than one car at the junction such that the decision making code as we described in section above will be removed. The reactive rule for traffic light will be added.

```
trafficLight(coordinate(X, Y), Color, FacingDirection,Street) at T->
  opposite(Color,Color2),
  Reminder = mod(T,40),
  Reminder == 0,
  changeTrafficLight(coordinate(X, Y),Color2) at T.
```

the above code means every 40 time cycle the color of the traffic light will be opposite (from red to green or from green to red).

```
% updating the traffic light
updates(changeTrafficLight(Place,Color) , trafficLight(PlaceIns,
    OldColor, _, _), trafficLight(PlaceIns, Color, _, _))<-
  % testPrint('traffic light as been updates'),
  Place == PlaceIns.
```

The action of traffic light need to be added to change the state of the traffic light

```
<- step(Vehicle, NextPlace),
  location(Vehicle, coordinate(X, Y), Direction1),
  trafficLight(coordinate(A, B), red, Direction2, Street),
  X==A,
  Y==B.
```

Restriction of step action need to be added since vehicle can not go forward when the traffic light is red

3.3.6 Cross junction with traffic light

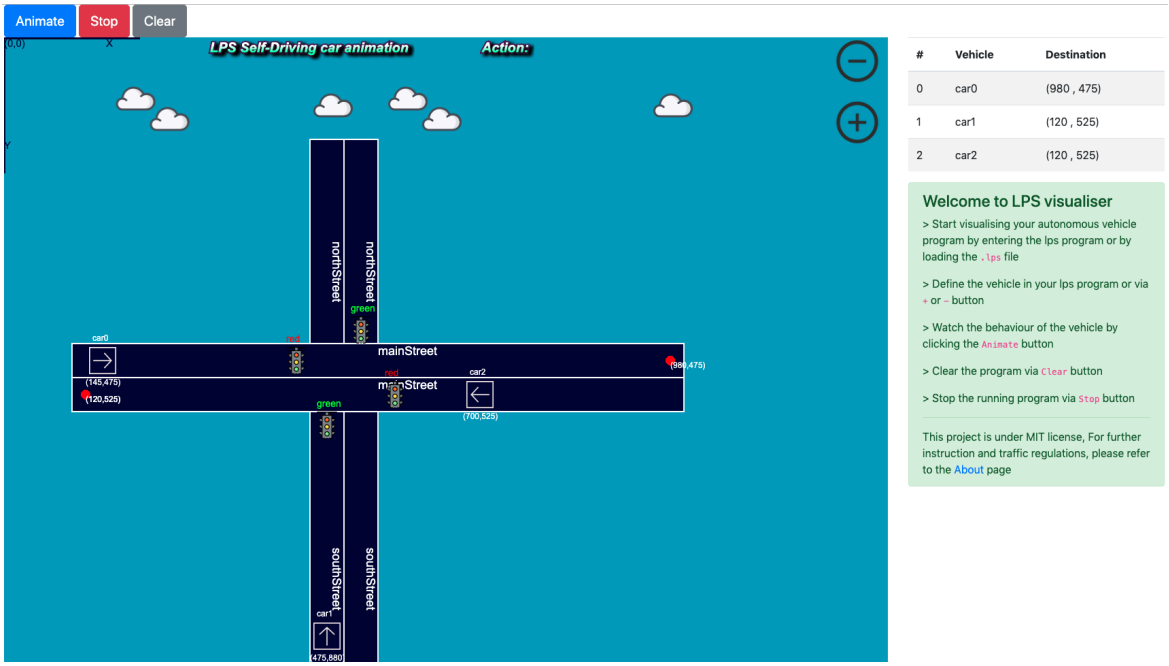


Figure 18: In the case of the cross junction there are still 4 junction block in total so the reactive rule and the restriction of the step action is the same as the previous two double lane T junction road. No more reactive rule is introduced here.

3.3.7 Traffic with restart feature

To visualise traffic flow better, a restart feature is design to putting the vehicle back to original point instead of stop at the arriving location. As we do not have information of the initial fluent so the start location need to be stored as a facts.

```
startLocation(car0, coordinate(145, 475), eastward),
startLocation(car1, coordinate(475, 880), northward),
startLocation(car2, coordinate(700, 525), westward),
```

For the traffic light with restart feature

```
% if the car is stopped and it has a start location.
stopped(Vehicle) at T ,startLocation(Vehicle, _, _)->
  restart(Vehicle) at T.

terminates(restart(Vehicle),  stopped(Vehicle)).
```

```
6 initiates(restart(Vehicle), moving(Vehicle)).
7 updates(restart(Vehicle) , location(VehicleIns, OldPlace,
     OldDirection), location(VehicleIns, NextPlace, NewDirection))<-
8   VehicleIns == Vehicle,
9   startLocation(Vehicle,NextPlace,NewDirection).
```

The new reactive rule is define as above saying whenever the car in stopped fluent it will restart fron the starting point. Hence a new action is also need to be written as follow:

3.3.8 Other Traffic road interface

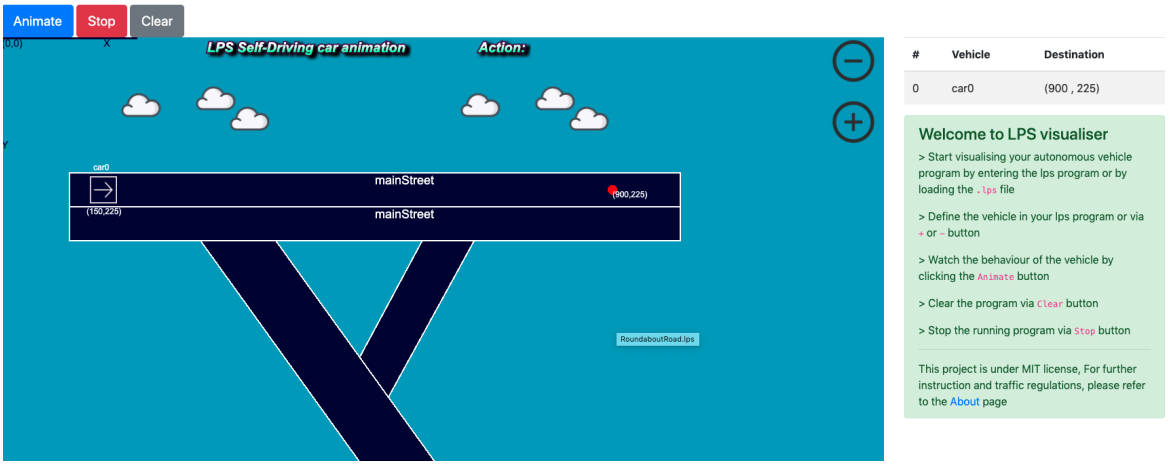


Figure 19

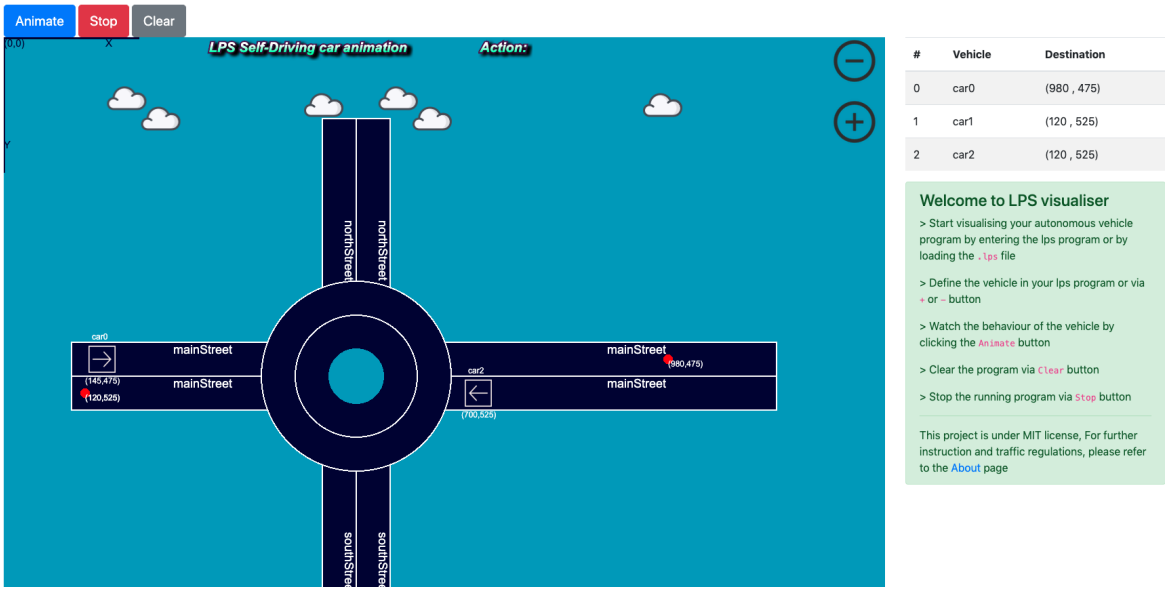


Figure 20

Fig. 20 and Fig. 19 are the The interface of define the roundabout and the Y shape junction is also defined in the project. However due to the limitation of the direction

it only restricts to north south west and east. Vehicle is not able to be similitude in Y shape road and roundabout.

??

3.3.9 Passing through

Figure 21

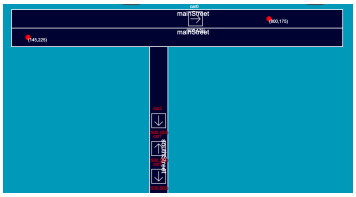


Figure 22

Figure 23

while two cars are driving towards each other (within a safety distance) we can assume two car can squeeze by each other, indicate by turning the text and location into red. After this stage the colour of the vehicle will turn back to it origin colour.

3.3.10 Overtaking

The car need to perform overtaking so different speed need to be assign to each car as a fluent. velocity(car1,5). meaning car1 has speed of 5 pixel per running cycle.

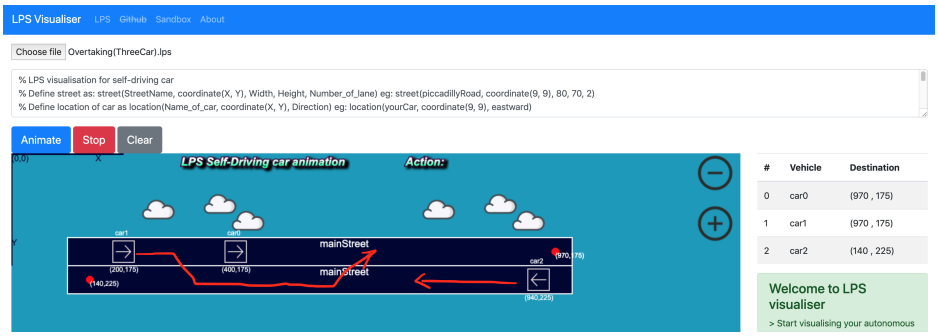


Figure 24: as describe in the background (section?) the car need to check if there another within its overtaking route if the overtaking route is clear the overtaking can perfume overtaking. If not car will follow in the behind of the car. As the example above, car1 will perform the overtaking because car2 will not collide with car1 during the overtaking action.

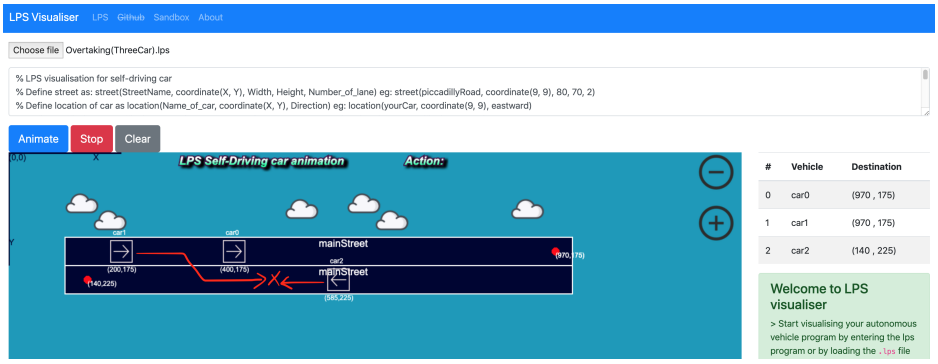


Figure 25: In the example above car1 will not perform the overtaking

```
1             collisionPossible(Vehicle, Vehicle2) from T1 to T3,
2   Vehicle != Vehicle2,
3   velocity(Vehicle, S1),
4   velocity(Vehicle2, S2),
5   S1 > S2,
6   blockedRoute(Vehicle,Vehicle2) at T1 ->
7   overtaking(Vehicle,Vehicle2,T1,T3).
8
9 overtaking(Vehicle,Vehicle2,T1,T3) <-
10  location(Vehicle, coordinate(X2, Y2), _) at T1,
11  NewY = Y2+50,
12  NewLocation = coordinate(X2, NewY),
13  aheadClear(Vehicle,Vehicle2,Vehicle3),
14  not goalTooNear(Vehicle),
15  step(Vehicle, NewLocation,T2, T3).
16
17 navigateBack(Vehicle,T1,T3)<-
18  location(Vehicle, coordinate(X2, Y2), _) at T1,
19  NewY = Y2-50,
20  NewLocation = coordinate(X2, NewY),
21  step(Vehicle, NewLocation,T3, _).
```

```
22
23 goalTooNear(Vehicle)<-
24   goal(Vehicle,coordinate(A1, B1)),
25   location(Vehicle, coordinate(X1, Y1), _),
26   Diff = abs(A1-X1),
27   Diff < 300.
28
29 not collisionPossible(Vehicle, Vehicle2) from T1 to T3,
30   velocity(Vehicle, S1),
31   velocity(Vehicle2, S2),
32   location(Vehicle, coordinate(X1, Y1), Dir1),
33   location(Vehicle2, coordinate(X2, Y2), Dir2),
34   Dir1 ==Dir2,
35   X1 > X2,
36   Y1 > Y2,
37   S1 > S2 ->
38     navigateBack(Vehicle,T1,T3).
```

LPS code of overtaking from upper lane to the lower lane is shown as above. If the front car has a low speed and is blocking the way them perform overtaking event. If the ahead route is also clear and the goal is far away then overtaking will be successful performed otherwise fail. 50 is the street width and 300 is the clearance distance setting as for convenience. However in reality the clearance should decide depends on the speed of the car.

It is clear you have completed lots of implementation, but it needs a better organisation. If it is possible to give the expected norms before the co
will put something together for one of the scenarios and send it to you.

3.3.11 Merging multiple cases into one

3.4 Front-end logic design and animation

Yes, not a section!

This chapter I will introduce the underlying front-end side of LPS visualiser. Which include three main part. Part one is the animation part which is mainly about the PIXI canvas. The second part is the link between the animation and data (fluent/-facts) how is the data parsed. The third part is about the layout design using twitter bootstrap and other open source toolkit.

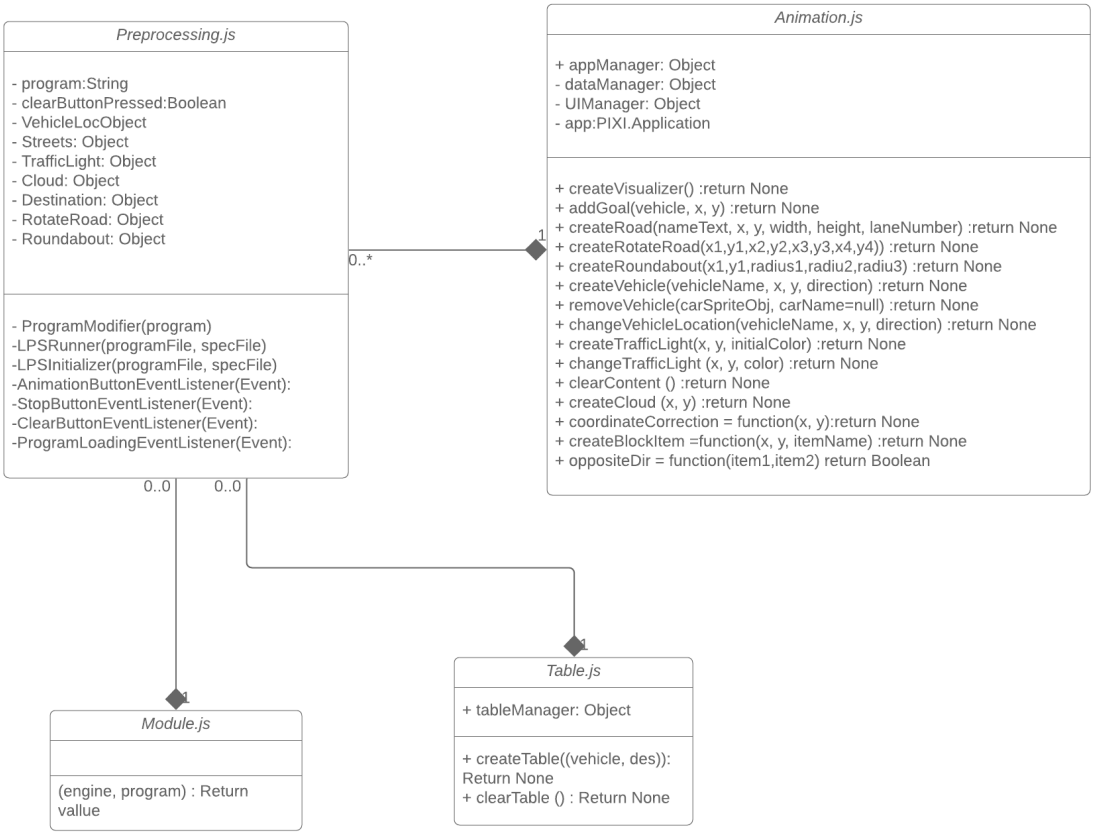


Figure 26: The class diagram of the front-end side is structured as above. The detail explained will be separated described below.

3.4.1 Animation.js

introduction The Animation.js create the PIXI canvas application for animation and read the changing fluent in LPS program simultaneously. To achieve this, a concurrency model need to be used. Animation.js use a range of data structure such as AppManager to maintain different list of object, UIManager to maintain a range of UI component.

structure The structure of the animation is showed as an activity flow chart. The animation.js will run once the user is on the animation page. Due to the DOM structure introduced in Section? The PIXI application will be created as soon as the page is loaded. The different event happened will fire different function. (1) is the creation of the PIXI application. PIXI has a tree data structure every component play a child node role in the whole application whenever a component is created it need to be append into the application tree.

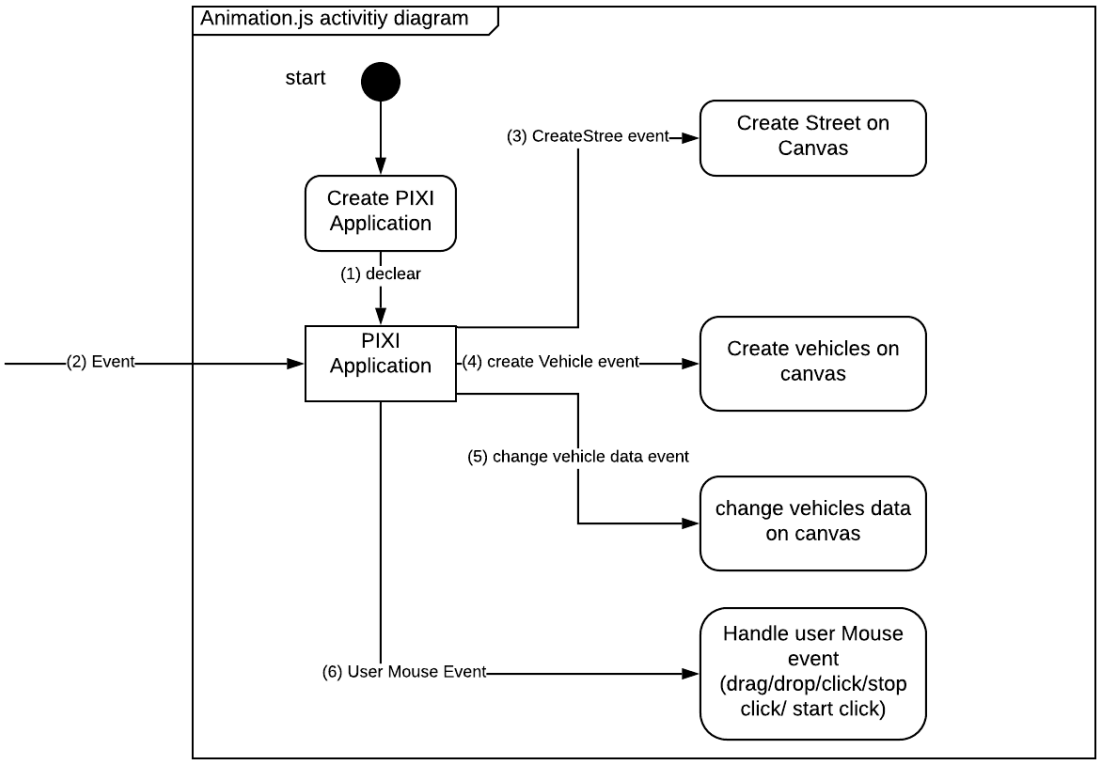


Figure 27

(1) Describe about the creation of the PIXI application

(2) Is different event whether is the event of the changing state of fluent or the user mouse event

(3) Handles the street creation. When street predication as a fact in LPS program is read into the LPS interpreter.

(4) Handles the creation of the vehicle When vehicle predication as a fluent in LPS program is read into the LPS interpreter.

(5) Handles the information changing of the vehicles, including direction, location, speed while the run-time of lps program. State of location predication will be detected by the engine and the PIXI application handles it by change the location on canvas accordingly.

(6) Handles the user event while includes the clicking event (click on car to change direction, click the stop button, click the animation button). Drag event, drop event(drop in deletion, drop on street). Some global information might change depends on the action that user performs.

Much better - maybe because you designed it all from scratch :-)

Figure 28

class fields There are four key components for Animation.js

Firstly, the appManager maintain the fluent entered by user such as vehicle, street, light, blocked item and so on. DataManager maintain the response used to detect user whether drop and drop the vehicle or just click on the vehicle to change the direction. Car counter is used here as an Integer type also used here

functionality createVisualizer Initialise the canvas, all the buttons, title text, define the base graphic

addGoal = function(vehicle, x, y) adding goal tag onto the map

createRotateRoad = function(x1,y1,x2,y2,x3,y3,x4,y4) This field will create the rotated road depends on the user defined

reateRoundabout = function(x1,y1,radius1,radiu2,radiu3) This field will create the roundabout

createVehicle = function (vehicleName, x, y, direction) This field will create the vehicles with name, location and direction. removeVehicle = function(carSpriteObj, carName=null) Will remove the vehicle if user drop the vehicel in the deletion area

changeVehicleLocation = function (vehicleName, x, y, direction) this field will modify the child in app

createTrafficLight = function(x, y, initialColor) this field will create the traffic light based on the location

changeTrafficLight = function(x, y, color) this field will change the traffic

clearContent = function ()this method will clear out the street and vehicle information * the basic canvas will remain the same (coordinate axis and the title text)

createCloud = function(x, y) this method will create the cloud and animate the cloud in the canvas

coordinateCorrection = function(x, y) correct the coordinate on to the street. use might place the car onto other part rather than the road

createBlockItem =function(x, y, itemName) When there is a blocked item placed by user/program the front end will show up a cross

oppositeDir = function(item1,item2) This part is checking all the adding cars that is there is any car that is opposite to each other (same horizontal or vertical level)

3.4.2 preprocessing.js

introduction The preprocessing.js handls the user input (except the input on PIXI canvas) such as mouse click the animation button, load the LPS program. Also prepossessing parse the useful information passed into PIXI app, such as vehicle location, street informaton. To achieve this, Document Object Model is introduced here.

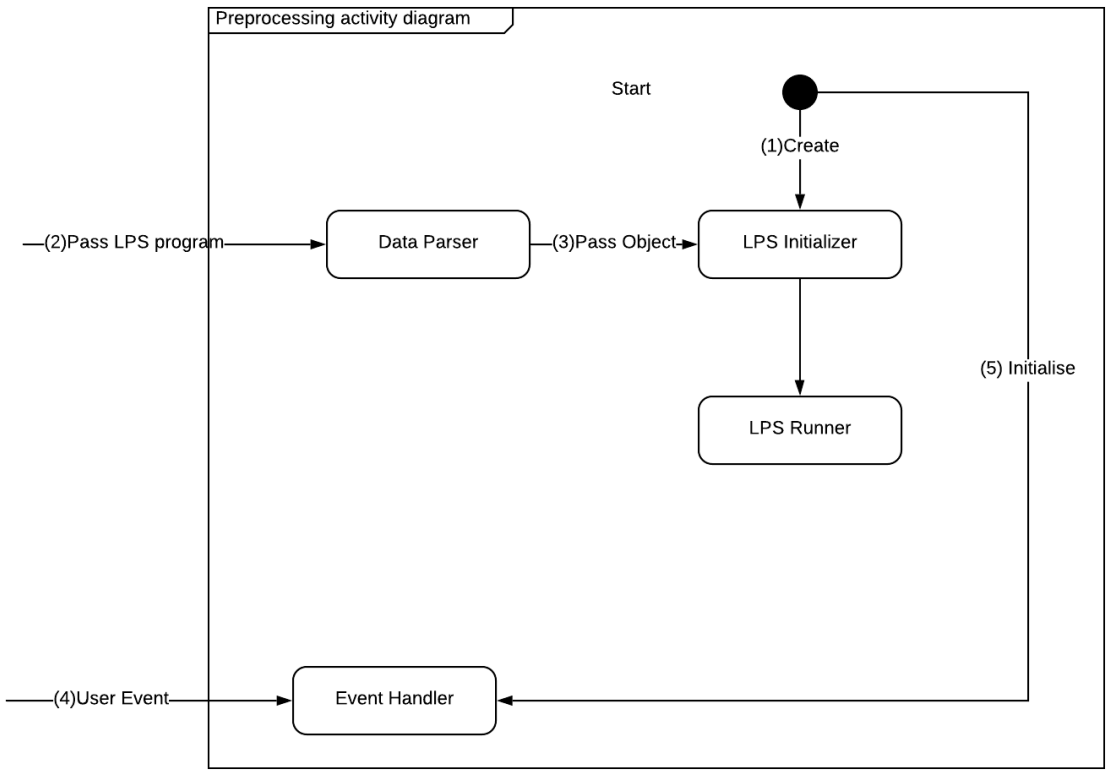


Figure 29

structure The diagram shows the activity diagram of the preprocessing.js. (1)The web page will create a LPS initialiser when user passed LPS program into the class. (2)(3)The passed LPS program is stored as string value so it need to be parsed into different object. The data which is parsed including the different road traffic light, vehicles and so on. Once the LPS program is passed in and parsed. LPS initialiser will call the animation class to create canvas vehicle etc. (4) The preprocessing.js class also handles the user input using Document Object Model of JavaScript. Such as user load the LPS program, user type in the text area. In Event handler a program modifier is also needed in case the user has pre-modified the program before running the interpreter. For example, user loaded the LPS program and user change the direction of the vehicle via PIXI canvas (LPS program has not been modified), the program modifier need to catch that modification such that the canvas has same information as LPS program.

Document Object Model (DOM) The Document Object Model is a programming API for documents. The object model itself closely resembles the structure of the documents it models. For instance, consider this table, taken from an HTML document: the change on the html can be easily detected and change via JavaScript and handle event accordingly. For example, when user load the LPS program this event will be handed and the LPS program will be parsed as string text into a text box area. By define JavaScript class as the follow structure we can also avoid the

ordering issue [17]

```
1 (function (window) {
2
3 })(window);
```

Figure 30

functionality

3.4.3 Demonstration Website design

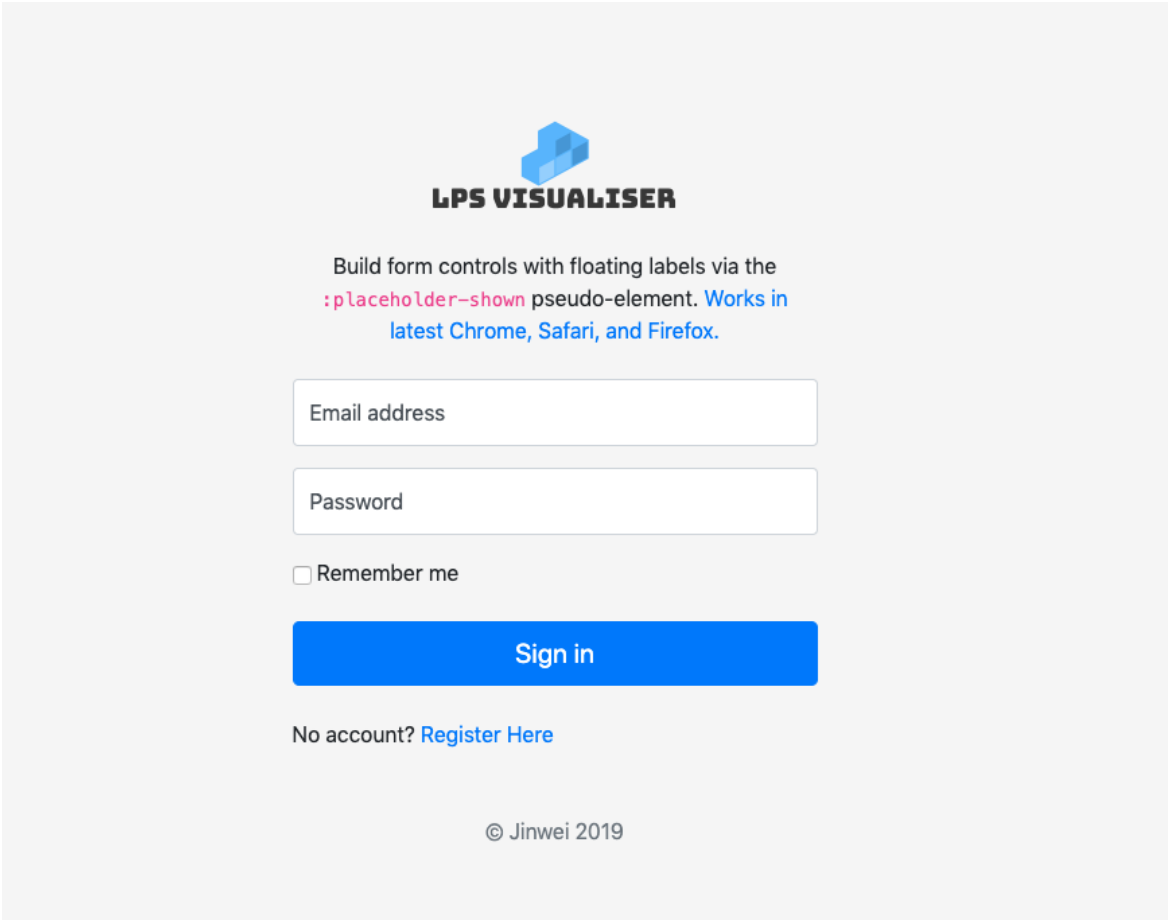


Figure 31: the login page

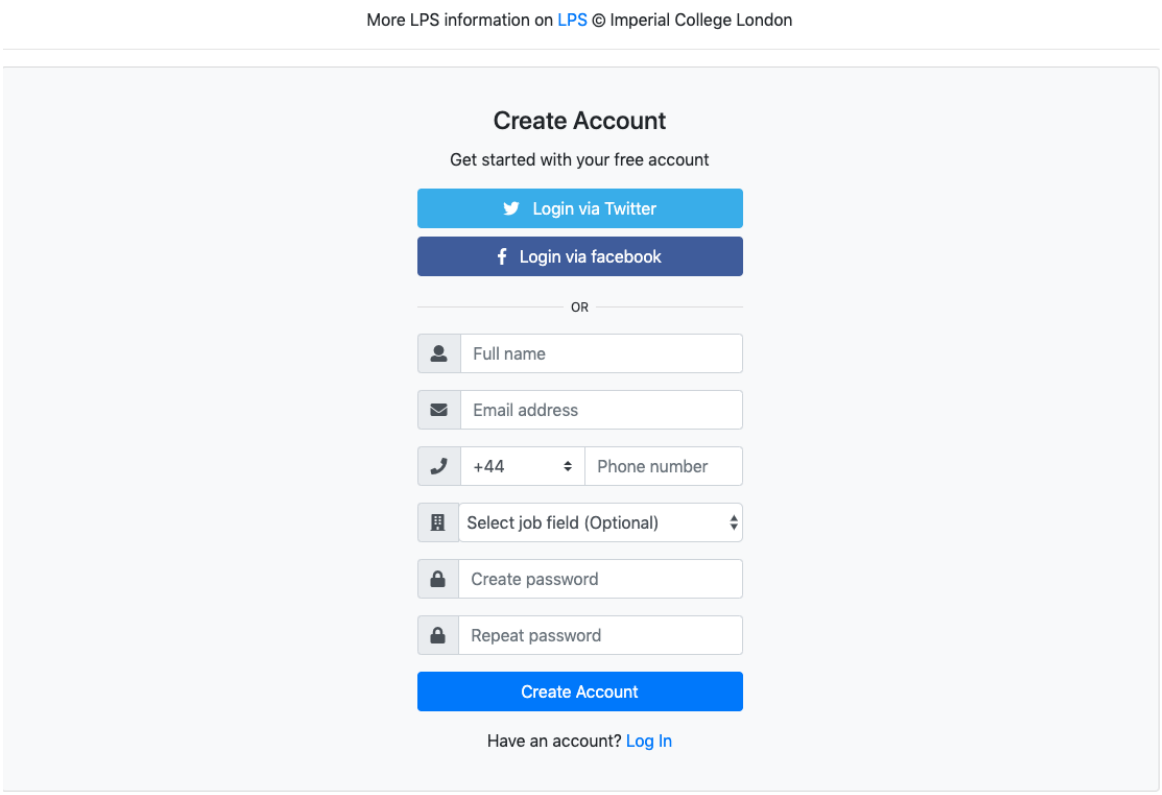


Figure 32: the registration page

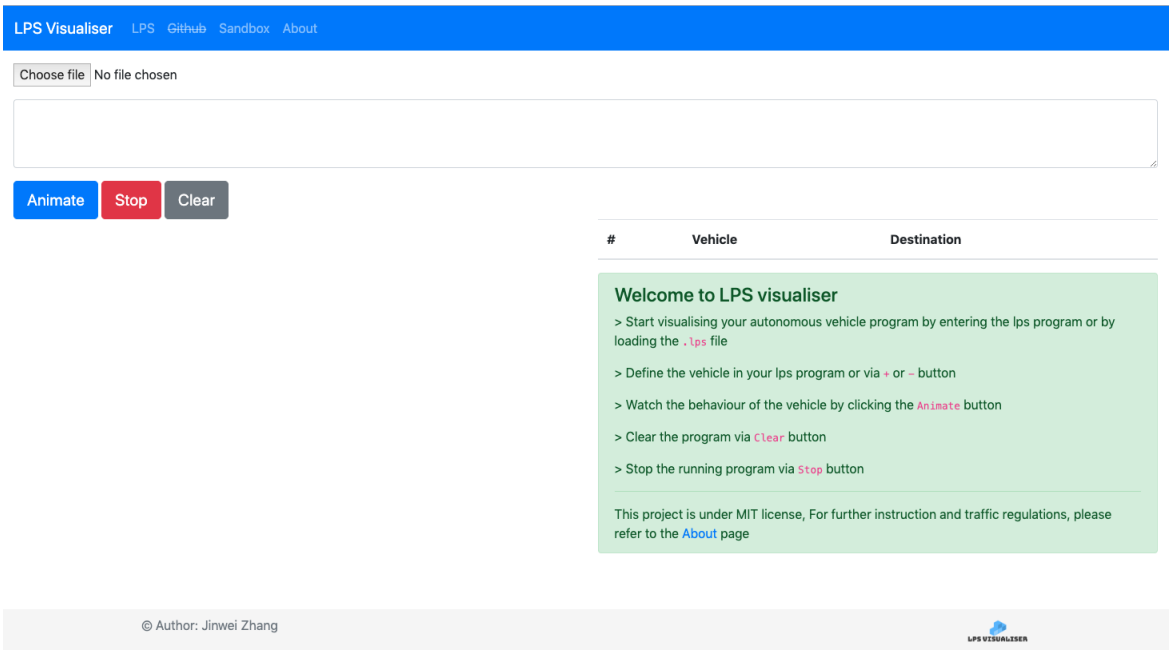
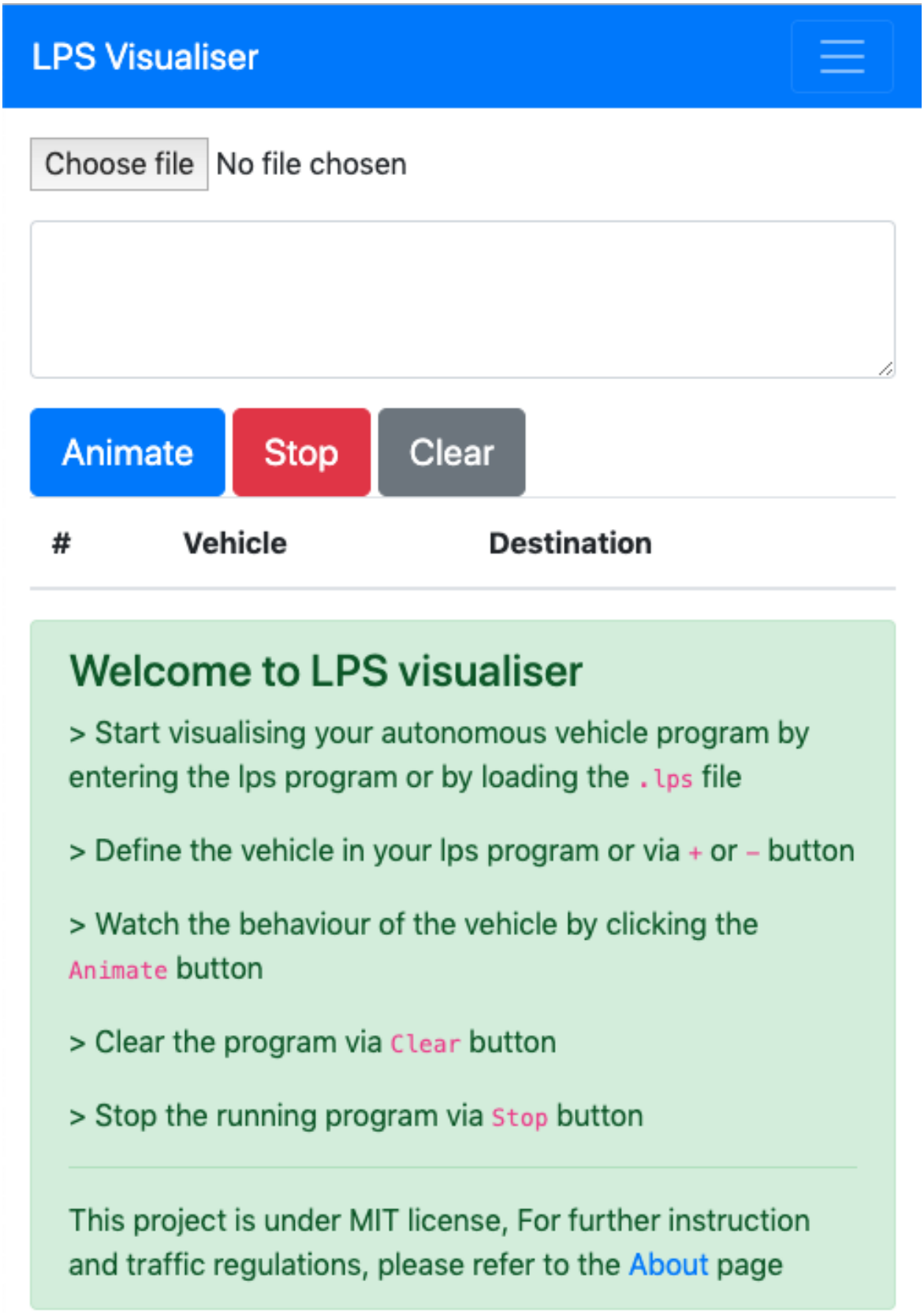


Figure 33: the main page by using boostrap it is also mobile friendly.


LPS Visualiser
Choose file
No file chosen
Animate
Stop
Clear
Vehicle Destination
Welcome to LPS visualiser
> Start visualising your autonomous vehicle program by entering the lps program or by loading the .lps file
> Define the vehicle in your lps program or via + or – button
> Watch the behaviour of the vehicle by clicking the Animate button
> Clear the program via Clear button
> Stop the running program via Stop button
This project is under MIT license, For further instruction and traffic regulations, please refer to the About page

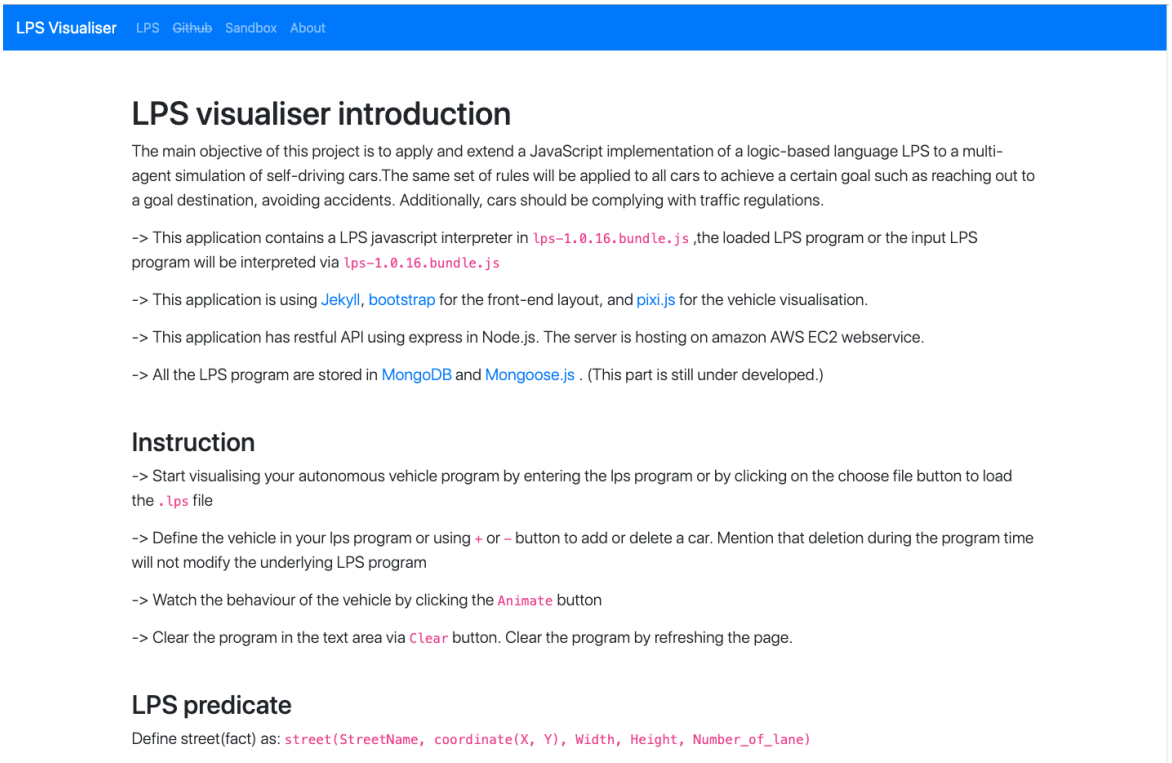


Figure 35: about us page. This page give a brief introduction of how to run the program and the technology involved

API and open source toolkit boostrap Jekyll is used in login page and register page. Font Awesome icons and css are used in register page.

3.5 back-end hosting

This chapter will introduce about the back-end of developing. The reason why back-end is introduced here is because website can be host and be accessed by everyone. Hence more function can be introduced here such as host LPS program in database, addition of LPS program into data base via user interface etc.

3.5.1 back-end hosting

introduction The back-end of the project is host on AWS server using express framework.

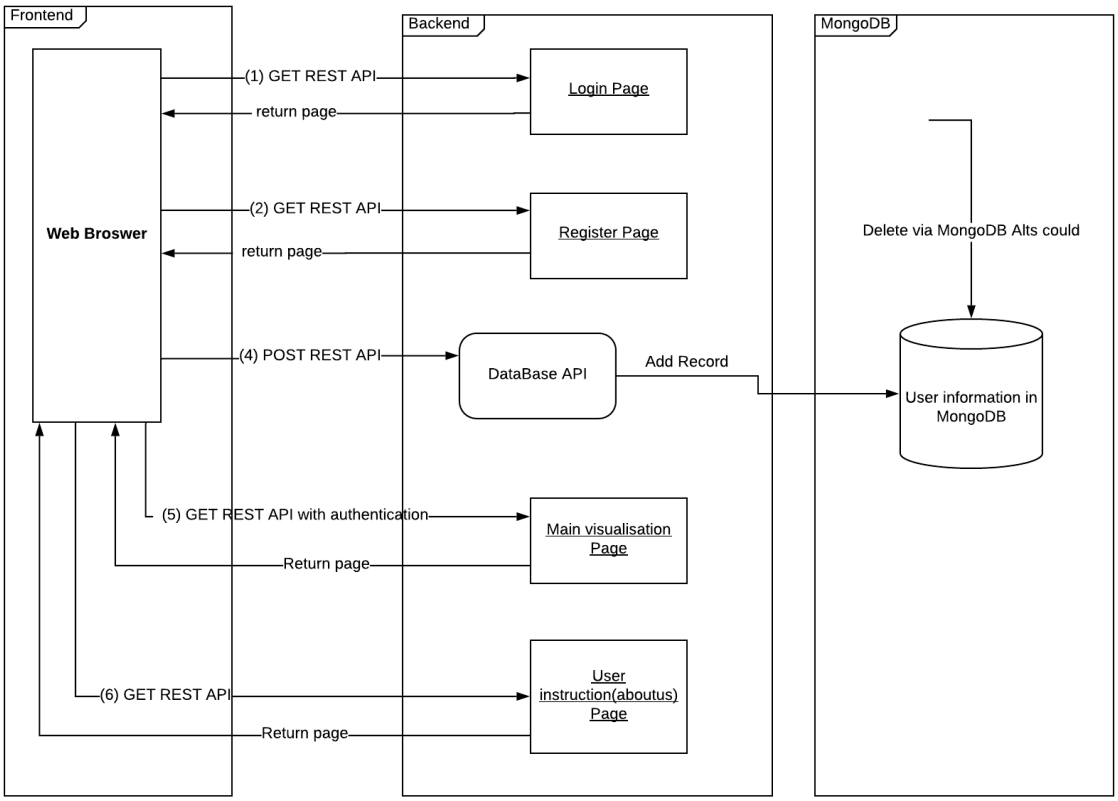


Figure 36: about us page. This page give a brief introduction of how to run the program and the technology involved

structure

Express.js

MongoDB and mongoose User information is parsed into userSchema and passed into MongoDB via POST REST API. User can view and manage the data in MongoDB alts cloud.

```
const UserSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    phone: {
        type: Number,
        required: true
    },
    job: {
        type: String,
```

```
16          required: false
17      },
18      password: {
19          type: String,
20          required: true
21      },
22      date: {
23          type: Date,
24          default: Date.now
25      }
26  });
```

Passport.js User authentication need passport.js to handle. Since the login page will catch a user input username and passport, passport.js is used to check against the database whether the user information is valid.

AWS EC2 server ?

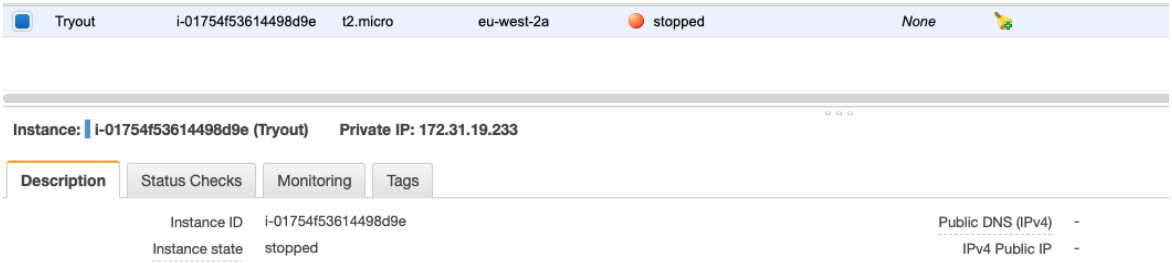


Figure 37: The server is EC2 instance, the server will provide a public IP(IPV4) once hooked with the port number, the website is ready to go online.

4 Evaluation

4.1 Usability

Easy to use up to an educational level. People have no programming experience should be able to program in LPS and visualise the code after going through the tutorial provided. LPS visitation should have well documented code.

If these are meant to be various desired criteria then itemise them in one subsection as requirements. You can then go through each one and comment on why you have achieved it, how you know you have achieved it,perhaps why it has not been completely achieved etc.

4.2 Efficiency

Efficiency means the code should be efficient to development also means low coupled and low coherence. Program should be hosted on a server and running in a reasonable amount of time. Also it should be efficient to be extended in the further.

4.3 Re-usability

As we have describe the problem about the previous visualization tool LPS.studio and the lps web version. This project should be easy to extend. All the back-end development should be based on the lps.ls. Because code for LPS.js is not documented so development might be too rushed to documented the code, hence there are too many lengthy function and useless variable and unnecessary function call-back etc. LPS visualization should keep the code as simple as possible also well documented. Testing should be formed to testing some key boundary cases moreover the unit testing should be written down.

4.4 Testability

The testing function which has already implemented are not useful for this visualisation project. Since all the testing is testing the class types, none of the test function is about logic and functionality. Also, the most of the test files are aiming to check whether there is the specification file generated using lps-generate-spec function which makes no sense to LPS visualization.

Here you could say rather more, giving some examples of typical test files - eg perhaps for one of the scenarios for instance.

5 Conclusion

5.1 challenges faced

5.2 Further work

TASK NAME	START DATE	DUE DATE	DURATION	% DONE	DESCRIPTION	SPRINT/MILESTONE
Task 1	03/06/2019	07/06/2019	4		Background report	Phrase0
Task 2	08/06/2019	13/06/2019	5		LPS.js to front end data entry	Phase 1
Task 6	14/06/2019	20/06/2019	6		Front-end design	Phase 2
Task 7	21/06/2019	27/06/2019	6		simple case with data	Phase 2
Task 10	28/06/2019	04/07/2019	6		complex case with data	Phase 2
Task 11	05/07/2019	18/07/2019	13		Back-end	Phase 3
Task 12	18/07/2019	01/08/2019	14		Back-end	Phase 3
Task 13	02/08/2019	15/08/2019	13		Front-end and back end connection	Phase 3
Task 15	16/08/2019	22/08/2019	6		Back-end	Phase 4
Task 16	23/08/2019	29/08/2019	6		Final report	Phase 4
Task 17	30/08/2019	06/09/2019	7		Final report	Phase 4

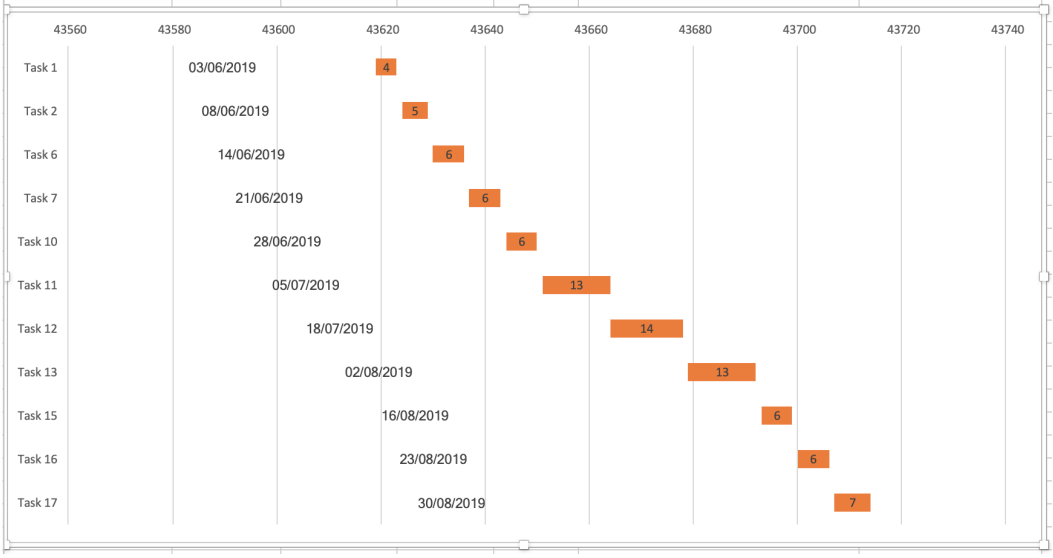


Figure 38: Gantt chart plan

This can go in an appendix or leave out!

Fig. 38 shows the plan that was made to achieve the goal. It was divided into 5 phrases. First of all, a background report need to be written to formalise all the background and the work that I have already done. The first phrase is the background report writing. Second phrase is the front end design and implementation. Third phrase is the back-end design. the fourth is to write the final report. Mentioned that the plan may various from the real-life due to some unexpected risk.

References

[1] Robert Kowalski and Fariba Sadri. *Introduction to LPS.* https://www.doc.ic.ac.uk/ rak/papers/LPS%20with%20CLOUT.pdf.

[2] Robert Kowalski and Fariba Sadri. *Towards a Logic-based Production System Language.* Technical Report, Imperial College London, 2010.

[3] Yong Shan Xian, Sam. Implementation of Logic Production System (LPS) in JavaScript. `https://github.com/lps-js/lps.js`, 2019. Online; accessed 04 June 2019.

[4] Robert Kowalski and Fariba Sadri. Reactive computing as model generation. *New Generation Computing*, 33(1):33–67, Jan 2015.

[5] P. A. Hancock, Illah Nourbakhsh, and Jack Stewart. On the future of transportation in an era of automated and autonomous vehicles. *Proceedings of the National Academy of Sciences*, 116(16):7684–7691, 2019.

[6] Allan M de Souza, Celso ARL Brennand, Roberto S Yokoyama, Erick A Donato, Edmundo RM Madeira, and Leandro A Villas. Traffic management systems: A classification, review, challenges, and future perspectives. *International Journal of Distributed Sensor Networks*, 13(4):1550147716683612, 2017.

[7] Lihua Zhao, Ryutaro ICHISE, Zheng Liu, Seiichi MITA, and Yutaka Sasaki. Ontology-based driving decision making: A feasibility study at uncontrolled intersections. *IEICE Transactions on Information and Systems*, E100.D:1425–1439, 07 2017.

[8] Martin Buechel, Gereon Hinz, Frederik Ruehl, Hans Schroth, Csaba Györi, and Alois Knoll. Ontology-based traffic scene modeling, traffic regulations dependent situational awareness and decision-making for automated vehicles. 06 2017.

[9] The Highway Code. Highway Code rules 159 to 203. `https://www.nidirect.gov.uk/articles/highway-code-rules-159-203`, 2019. Online; accessed 04 June 2019.

[10] A. Rizaldi and M. Althoff. Formalising traffic rules for accountability of autonomous vehicles. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 1658–1665, Sep. 2015.

[11] J. Nilsson and J. Sjöberg. Strategic decision making for automated driving on two-lane, one way roads using model predictive control. In *2013 IEEE Intelligent Vehicles Symposium (IV)*, pages 1253–1258, June 2013.

[12] Jan Wielemaker, Fabrizio Riguzzi, Bob Kowalski, Torbjörn Lager, Fariba Sadri, and Miguel Calejo. Using SWISH to realise interactive web based tutorials for logic based languages. *CoRR*, abs/1808.08042, 2018.

[13] Yong Shan Xian, Sam. LPS Studio. https://github.com/lps-js/lps-studio, 2019. Online; accessed 04 June 2019.

[14] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 307–325, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] Sukhjinder Arora. Understanding Asynchronous JavaScript. https://blog.bitsrc.io/understanding-asynchronous-javascript-the-event-loop-74cd408419ff, 2019. Online; accessed 13 Auguest 2019.

[16] maxogden. information about async javascript programming. http://callbackhell.com/, 2019. Online; accessed 13 Auguest 2019.

[17] Xah Lee. JS: Load Order. http://xahlee.info/js/js_executing_order.html, 2019. Online; accessed 16 Auguest 2019.

Appendices

Some Appendix