Imperial College
London

FINAL REPORT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

LPS-visualisation

Author:
Jinwei Zhang (CID: 01540854)

Date: September 1, 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].

As LPS describes a framework, it has already been implemented in different programming languages. LPS has been extended onto SWISH – an SWI-Prolog web application by Wielmaker et al. [3]. LPS has also been implemented in JavaScript by Sam Yong as an MSc project [4]. There is various animation engine using JavaScript on the market such as PIXI which is the fastest, most flexible 2D WebGL renderer.
With the mature implementation of LPS in JavaScript, achieving the function of animating LPS program become a more significant further work for the programmer to write LPS with a 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 system will be called LPS visualiser throughout the report. The same set of rules will be applied to all cars to achieve a particular goal, such as reaching out to a goal destination, avoiding accidents. Additionally, cars complying with traffic regulations. To achieve this objective, the following was required:

- Design an easy-to-use and well-documented front-end and back-end system.
- Design a suitable robust system for LPS visualization.
- Preparing different LPS program and cases for different traffic scenarios as this will be described in the background section

1.2 Requirement

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

1. Extensible
   The implementation should allow the developer and third parties to use or extend the features using the provided open window/API without having to modify the source code. For the contributor, the source code should be well documented such that the developer can easy to follow and contribute. The code should maintain a suitable engineering manner, such as low coupled and low coherence.

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

3. Efficiency
   The program should be hosted on a server and running in a reasonable amount of time. The asynchronous functions are considered to be used instead of synchronous using the promised feature in order to avoid unnecessary waiting time.

4. Testability
   Since the implemented program may have the issue of latency, wrong data structure parsing format etc, the testing code should be formatted as to test class types, program logic and functionality. Moreover, 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. It is structured in the following manner:

- **Section 1** is the introduction and a brief overview of the report.
- **Section 2** is about the background of this project, which consists of the detail about LPS, different traffic norms and different technologies that will be used in the project.
- **Section 3** is the detailed 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 and evaluation of its performance, particularly in the latency testing of goal-achieving strategies.
- **Section 5** concludes the report and recommends some future extension and research relevant to this project.

preconceived notions: The whole system is named as LPS visualiser. The report was written with the assumption that 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 LPS would start with a lower case letter. The variable names will start either with an upper case letter. The traffic norms are all under United Kingdom traffic laws. The reader should have a brief knowledge of LPS interpreter (LPS.js), JavaScript in web front-end programming and Node.js.

2 Background

2.1 LPS

LPS stand for logic production system "it combines production rules and logic program in a single logic-based framework". An operational and model-theoretic semantics is associated with production rules, and a database of facts extends the logic programs [2].
Production rules[2]:

1. Production rules are consist of three different kinds where are reactive rules, forward chaining logic rules and goal-reduction rules.

2. Reactive rules are typically ***condition*** $\rightarrow$ ***actions*** like the form of condition in logic. Such as: "If it is raining on Sunday $\rightarrow$ stay in the home."

   You should write this in logic as well. I think you can pick a better example, one from your traffic application.

3. Forward chaining logic rule is, for example, $DogX \rightarrow loyaltyX$ adds loyalty X to the working memory as a fact.

   Need brackets here

4. Goal-reduction rules are rules have general form of goal G and conditions C $\rightarrow$ add I as a goal. LPS then use backward reasoning like $G \leftarrow C, I.$ to update and manipulate the goal. The working memory contains both "real" facts as well as the goal facts. The "real" facts describe the current database state while goal facts describe some future desired state.

Do you use goal rules like this? if so, use an example from your traffic application. If not, perhaps simply say they exist but are not used in this project.

Production system [2]:

1. 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.

2. The working memory can be viewed as a model-theoretic structure

3. The language of the condition of a rule is generally a subset of first-order logic

Formally, McDermott and Forgy defines a production P_i[5] to be

$$P_i[C_1 \ldots C_n \rightarrow A_1 \ldots A_m] \tag{1}$$

where $C_1 ... C_n$ represents the conditions that need to be checked against the working memory. The $A_1 ... A_m$ are the actions to be executed once the conditions are met.

production system framework[6]:

You could almost start the background from here. What went before would be better in prose style rather than bullet points. Moreover, it would be better after the LPS description, in my view.

1. Logic production system framework is in the form of ***<R, L, D>***

2. **R** stands for Reactive Rules, it is in the form of $\forall X[$ antecedent $\rightarrow \exists Y$ consequent $]$

3. "Events include external events and an agent's own internally generated actions. Events are represented by atomic formulas including time and possibly other parameters".

   For an example, the variables T_1, T_2 and T_3 represent time, drive(Loc, T_2, T_3) represents a possibly complex (or composite) action performed from time T2 to time T3 and observe-goal-at(Loc, T_1, T_1+1). represents an external event that triggers the action:

   T2>T3

   normally comma between variables Loc and T1 (and similarly T2 and T3)

   $$\forall Loc T_1\,[\text{observe-goal-at}(\text{Loc}, T_1, T_1+1) \rightarrow \exists T_2 T_3\,[\text{drive}(\text{Loc}, T_2, T_3) \wedge T_1 < T_2]]$$

   The JS version: What's this?

   $$\text{observe-goal-at}(\text{Loc}, T_1, T_1+1) \rightarrow \text{drive}(\text{Loc}, T_2, T_3)$$

4. **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).

   (1) .....; (2) ....; (3) ... (or better to put them on separate lines)

   The clause below defines the composite action drive(Loc, T_1, T_3) in terms of composite action arrive(Loc, T_1, T_2) and actions terminate-moving-state(T_2, T_2+1)

   How is T3 related to T2? Is T3 =T2+1?

   $$\text{drive}(\text{Loc}, T_1, T_3) \leftarrow \text{arrive}(\text{Loc}, T_1, T_2) \wedge \text{terminate-moving-state}(T_2, T_2+1)$$

   The JS version: Looks the same?

   $$\text{drive}(\text{Loc}, T_1, T_3) \leftarrow \text{arrive}(\text{Loc}, T_1, T_2) \;,\; \text{terminate-moving-state}(T_2, T_2+1)$$

5. **D** stands for domain theory $\boldsymbol{D} = D_{pre} \cup D_{post}$. It is the precondition and the post condition of state transaction.

   For example, the following clauses define the post-conditions of the event observe-goal-at(Loc, T, T+1) and the action arrive(Loc, T, T+1): (Same as JS version)

   $$\text{initiated}(\text{moving-state-at(Loc)}, T+1) \leftarrow \text{observe-goal-at(Loc}, T, T+1)$$
   $$\text{terminated}(\text{moving-state-at(Loc)}, T+1) \leftarrow \text{arrive(Loc, T, T+1)}$$

Operational Semantics[6]:

You need to say where the quote comes from

1. The operational semantics (OS) can be represented by a non-terminating cycle. "It uses a destructively updated database and combines the features of a logic programming language, the database language with the reactive rules of production systems, and BDI agent languages, such as Agent-Speak ". The external events and the agent's executed actions are merged, and the state is updated destructively in OS.

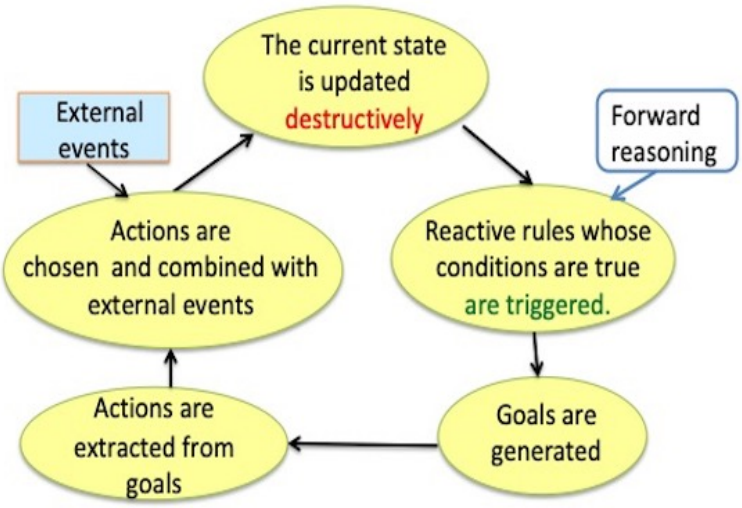


Figure 1: Cycle: When the Reactive with all true conditions are triggered, the OS will maintain the current state S_i as well as a goal state G_i. A goal state G_i is a set (or conjunction) of goal trees. When the antecedent becomes true by forward-reasoning, a goal tree is generated. Every node in a goal tree is a goal clause which represents an alternative way of solving the goal clause. Through updating the current state by performing action and external event, all the goal trees must eventually need to be reduced to true by in order to solve the computational task.

To be honest I don't think this caption is enough. What does a goal tree look like, for instance?
As I suggested in the previous version, you need to work through a very simple example (say with one or two rules maximum and perhaps one or two pre/post conditions.

2.2 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 [7][8]. 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. There are several different road layout, which have norm of behaviours to simulate real traffic. They are roundabout T-junction road ,cross-junction road and overtaking. These norms will be introduced in detail below.

On average?

Put in italics so it stands out.

2.2.1 Roundabouts

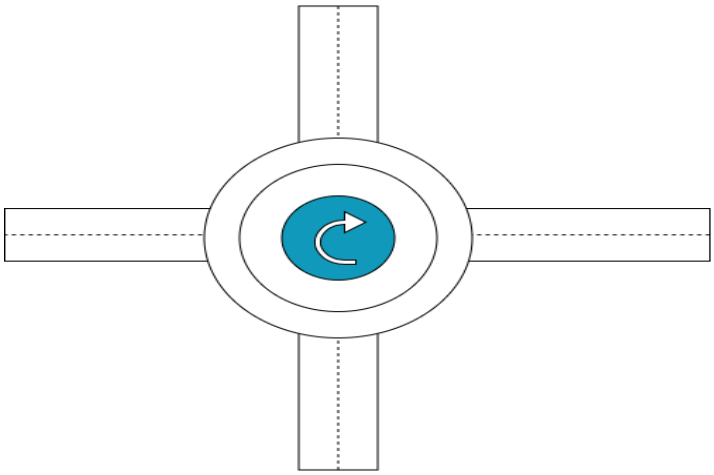


Figure 2: Roundabouts

Why is the text in separate chunks? Join it into one paragraph.

a number of junctions meet
Roundabouts are traffic circle where number of junctions meets from different direction in a one-way circular road.

They are (or .... country)
Roundabouts are designed to keep the traffic flow in clockwise or counter-clockwise depends on the country. In this project UK standard which is counter-clockwise will be implemented and demonstrated. the UK

The circular road is a street and maybe made up of one or more lanes.

Though multiple lanes is complicated. You can make it easier by assuming a single lane

The traffic entering the roundabouts must give way to traffic already on the roundabouts approaching from the right.

Overtaking
overtaking is prohibited on roundabout.

2.2.2 Uncontrolled T-junction

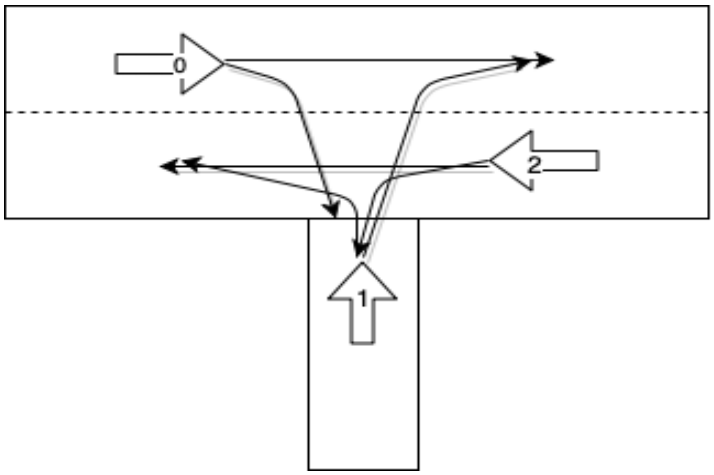


Figure 3: This is a uncontrolled T junction graph, re-sketched from paper "Ontology-Based Driving Decision Making" Fig.15 [9]. As showed in the picture, the solid line represent the intention of each car. In this particular graph, there are 6 different cases that might cause collision. (shown in detail in Figure 4)

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

Again, make one paragraph

It may be T-Junction may be uncontrolled or controlled by traffic light, however, cars need to either 'Give way' –give priority to traffic on the main road. or 'Move on' based on the intention that the nearest car is making. comma

If the road is uncontrolled with no sign, according to the simulation experiment in research"Ontology-Based Driving Decision Making" [9]. There are six possible paths overall and two maximum upcoming collision for the vehicle0 as shown in fig. 3. Not a sentence

The vehicle need to follow the SWRL Right-Of-Way rules [9] 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 vehicle0.

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.

If the turning into narrow road is turning left why should it wait?
I think what you mean is a narrow road which does not have room for two cars except at certain places, called "cutouts"?

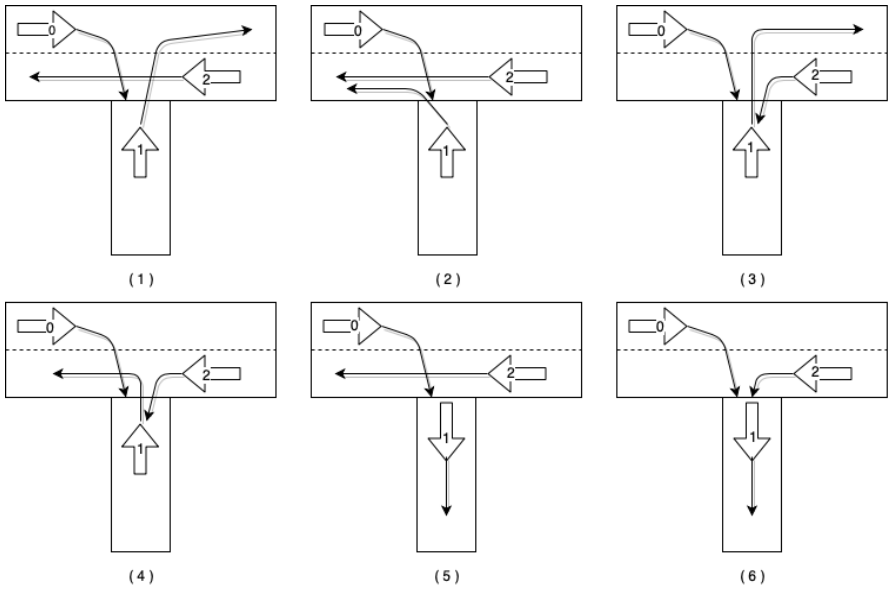


Figure 4: shows a detailed version of 6 different decision making in uncontrolled T-junction from the paper "Ontology-Based Driving Decision Making"[9]. In first four cases, it shows the vehicle0 should give way to both cars because one vehicle is driving straight and the other vehicle is driving out from narrow road to the wider road.

2.2.3 cross-junction

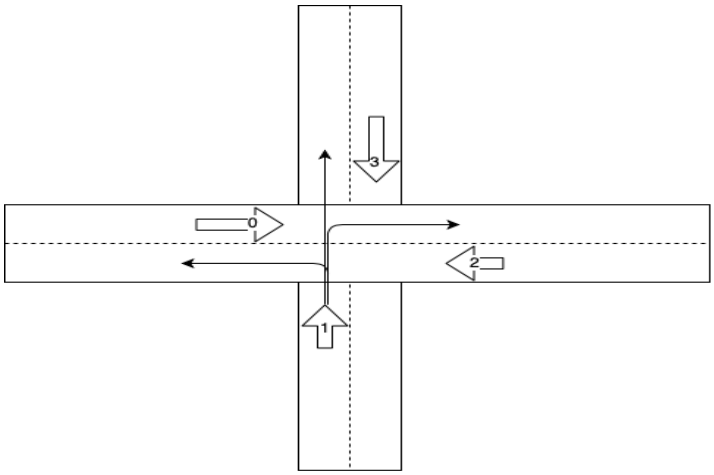


Figure 5: Cross-junction is the place where two road cross with a perpendicular angles as shown in Fig. 5. Cars must give way to traffic on the main road when emerging from a junction with broken white lines across the road.

Each car has 3 possible move (go straight,turn left, turn right.). For 4 cars in a junction there are 3*4 = 12 cases scenario which make decision very hard. However, to be more realistic, if we only consider the nearest three cars. any collision in the cross junction can be convert to one of the previous T junction case.

Actually, if a car turning right has to give way to a car in opposite direction turning left or going straight ahead, then to turn left only need worry about car coming from right; to go straight ahead must worry about cars from right or left; to turn right worry about cars coming from left or right or going straight on (from opposite direction, but the first rule deals with that); hardest case is when cars in opposite directions both want to turn right!! One of them must give way, usually or CRASH!

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

Figure 6: In the research of "Traffic Scene Modeling" [10]. A sample set of rules for uncontrolled cross junction in SWRL can be written as above. When vehicle is on a uncontrolled junction and intention c1 and intention c2 are conflict. One car has to give way to another car based on further predication and conditions. Similar rules will be formalised in LPS in the implementation part.

no full stop - same sentence

2.2.4 Junctions controlled by traffic lights

Each road will have a traffic light to control the traffic flow. Vehicle will not make decision based on other cars intention or making less decision.
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 [11].
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.
In this project stop line are replaced by traffic light. Vehicle will stop before the traffic light instead of the stop line.

by a traffic

2.2.5 Overtaking

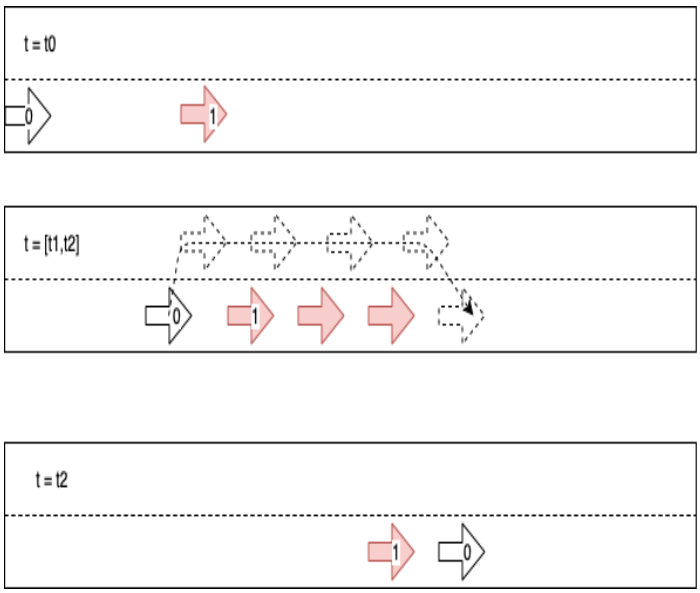


Figure 7: overtaking case [12]

When a vehicle overtakes

Fig. 7 is an illustration of overtaking scenario. When vehicle overtake another traffic participant, the other lane must be free of any traffic occupancy. The ego vehicle in Fig. 7 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 as shown in t = [t1,t2][12].

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 [13]. The mix of logic and integer linear inequalities should be formed so that the collision can be avoided

Again make into one paragraph

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

2.3 Technology

2.3.1 Swish visualisation

have been investigated before implementing LPS visualiser

Different visualisation technologies has been experienced before implementation LPS visualiser so that the most suitable tool can be chosen to use. LPS has been translated in SWISH language which is implemented in SWI Prolog[14]. SWISH has the feature of easy to code and easy to understand. However, swish is difficult to visualise a large amount of agents. Function of defining object including street, vehicles and animation code is lengthy and hard to understand. An example of visualising one vehicle object is shown in Fig 11. Due to the reason mentioned above, the idea of achieving visualisation through SWISH was tried but abandoned at the early stage.

2.3.2 LPS studio

LPS Studio[15] is an interactive storytelling tool for visualising Logic Production System (LPS) programs using the lps.js interpreter on top of using Electron and Angular frameworks. LPS studio allow different animation predicate such as roadEdge() to define road, lpsDefineObject() to define objects. If these predicate are defined in the LPS program and LPS studio are able to animate consequently. The example of using LPS studio are demonstrated in the implementation part in Fig. 12 and Fig. 13. However, different bugs has been found such as testPrint() command will terminate program if it is placed before action. Animation thread conflict with array, conflict with recursion etc. Moreover the p2p function is not working due to the problem of server not receive connection. Since LPS studio has some problems that have been encountered, and debug the not well documented code is time consuming and worthless, this project need to using other third part framework and be implemented from scratch.

Check English in this sentence.

2.3.3 LPS.js

LPS.js [4] is the JavaScript interpreter engine for LPS. LPS program is written as a JS understandable style in .lps file. It was implemented as a Msc student's project "Implementation of Logic Production System (LPS) in JavaScript" [4]. LPS.js is able to parse and run the written LPS program into JavaScript format. LPS.js has a bundle version and it is easy to hooked with web. The LPS visualiser use the idea of recording the state of the fluent run by LPS.js engine and implement the visualization part in the front-end web.
The problem of the previous work such as undocumented code, bug in peer-to-peer function, need to be considered during the implementation stage. Bugs that need to be avoid when implementing LPS visualiser.
Due to the problem of lacking a API for direct write LPS in JavaScript, LPS visualiser need to provide different parser to format the LPS code into JavaScript.

I don't follow this. First sentence says LPS.js can parse LPS into javascript. Do you mean that parsing and running are part of the LPS.js, but you cannot just do the parsing part?

2.3.4 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 run-time implement of lps.js. It is implemented as a node.js package under npm. It can be easy to install in linux and windows environment if node.js is installed. LPS Cli tool will be used to debug and test LPS visualiser.

2.4 JavaScript

JavaScript often abbreviated as JS, it is an interpreted high-level, scripting language that conforms to the ECMAScript specification. JavaScript, HTML, and CSS and are three core technologies of the World Wide Web. JavaScript is an essential part of web applications, it enables interactive web pages, and it is also the top language based on repositories on GitHub, as shown in Fig. 8. There are many third party rendering tool for animation written for web application in JavaScript. As this reason, JavaScript becomes the primary language in this visualisation project.

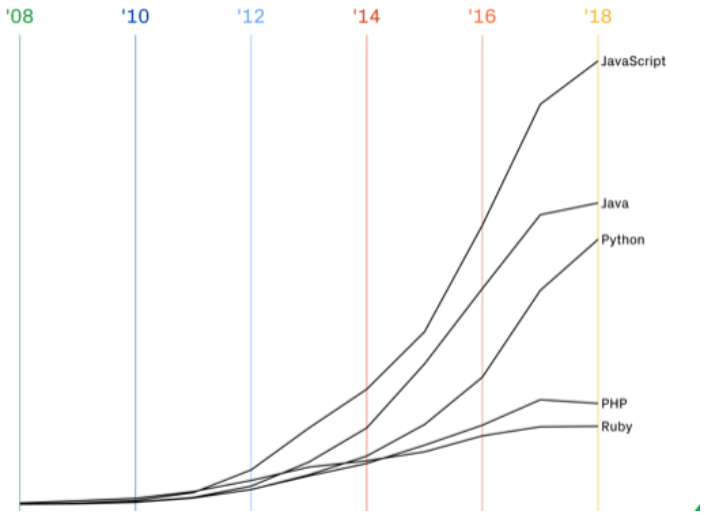


Figure 8: Top programming languages by repositories created, 2008-2018, JavaScript tops the list for the language with the most contributors in public and private repositories from the year 2009.

2.4.1 JavaScript asynchronous programming

English is unclear

JavaScript is a single-threaded programming language. It means only one thing can happen at a time [16]. However, running the LPS animation will blocking the main thread might and take some time to process this will make web page unresponsive. That is where asynchronous JavaScript is introduced. Using asynchronous JavaScript method such as callbacks, promises, and async/await can make the web application more responsive. It also allows users to perform other function such as stop the program, clear the program, delete the content while the LPS program is running. To understand how asynchronous in JavaScript works, The next paragraph will introduce a few topics related to it.

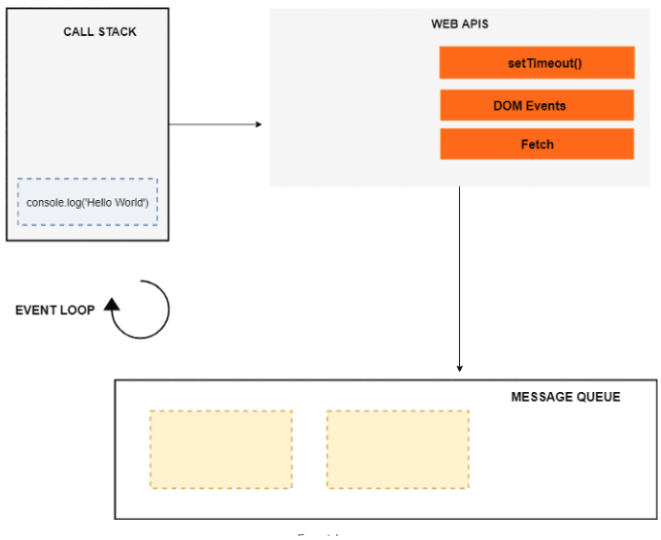
2.4.1.1 Execution Context An Execution Context is an abstract concept that holds information about the environment within which the current code is being executed [17]. Whenever any code is running in JavaScript, it is running inside an execution context. The global execution context is created before any code is executed. Whenever a function is invoked or executed, a new execution context gets created. Every execution context provides this keyword, which points to an object. The object belongs to the current code that's being executed.

2.4.1.2 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 has a single call stack. When it comes to asynchronous execution, asynchronous callbacks are introduced to understand the callback event loop and the callback queue(message queue/task queue) need to address. One case of using callbacks to parse LPS program into interpreter engine is demonstrated in

3.1. There are different type of asynchronous callbacks used in this project and the detail will demonstrated in the implementation part.

2.4.1.3 Event loop The Event loop will check the call stack to determine whether the call stack is empty. It will further review the message queue to see if there is any pending callback waiting to be executed if the call stack is empty. 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');
```



Unclear really, how this relates to code above.

Figure 9: The execution of the code: Firstly browser loads console.log('Hello World') is pushed to the stack and popped off the stack after it is finished. Secondly, networkRequest() is encountered, so it is pushed onto the top of the call stack. Following next, setTimeout() function is called, so it is pushed to the top of the stack. Mention that the setTimeout() has two arguments: (1) callback and (2) time in milliseconds. Since the setTimeout() need some time to be executed the timer of 2s will be recorded in the web APIs environment. The setTimeout() will finish execution and will be popped off from the call stack. After that, console.log('The End') will be pushed onto the stack, executed and removed from the stack when it has completed. Meanwhile, the timer in the web API has expired, so the callback is pushed to the message queue. Notice that the callback is not immediately executed, it is executed after the execution of console.log('The End') and that is where the event loop comes to the play.

2.4.1.4 The Event Loop In the previous example, 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 will further look into the message queue to see if there is any pending callback waiting to be executed.

2.4.1.5 DOM Events DOM (Document Object Model) events are actions that occur as a result of the user action when a web page has loaded or as a result of the state change of the elements of a DOM tree. Examples of HTML DOM events like user clicks the mouse on a button. DOM events can handle user event asynchronously. The event listener situated in the web APIs environment and listen for a specific event (such as click event, mouse input event, timeout expired) to happen. When that event happens, the callback function will then be placed in the message queue waiting to be executed.

2.4.1.6 Promise Callbacks being used repeatedly and irresponsibly by the programmer can be lead to a phenomenon called ”pyramid of doom” or ”callback hell”[18]. The callback hell code can become hard to maintain and flooded scope with irrelevant variables, making it difficult to debug and maintain. JavaScript ES6 introduced the concept of the job queue, also called micro-task queue. JavaScript "Promise" achieve it. The main difference between the job queue and the message queue introduced in the previous paragraph is that the priority of the job queue is higher than the message queue. It also means that promise jobs inside the job queue/ micro-task queue will be executed before the callbacks inside the message queue. For abbreviation, the event loop will first empty the job queue / micro-task queue before executing any callback in the message queue. For example:

```
1 console.log('Script start');
2 setTimeout(() => {
3   console.log('setTimeout');
4 }, 0);
5 new Promise((resolve, reject) => {
6     resolve('Promise resolved');
7   }).then(res => console.log(res))
8     .catch(err => console.log(err));
9 console.log('Script End');
```

The execution sequence will be : $Line1 \rightarrow Line2 \rightarrow Line5 \rightarrow Line9 \rightarrow Line7 \rightarrow Line3$

2.4.2 Node.js

Node.js was invented initially by Ryan Dahl in 2009, and It is an "open-source, runtime environment that executes JavaScript code outside of a browser [19]." Node.js allows developers to use JavaScript to write command-line tools, and it can also achieve server-side scripting. It can produce dynamic web page content before the user’s web browser requests the page.
As an asynchronous event-driven JavaScript run-time, Node is designed to build scalable network applications. Node presents an event loop as a run-time construct as we describe in the section "The Event Loop." In other systems, A blocking call will always be called to start the event-loop. Usually, this behavior is defined by a callbacks function at the beginning of a script, and the end starts a server through

another blocking thread call like "EventMachine: run()." However, In Node, there is no such start/end event-loop call. This behavior is similar to browser JavaScript — the event loop is hidden from the user. Node will jump into the event loop after executing the input script. It will exit the event loop when there are no more callbacks to execute.

I must wonder that all the above does not come with a citation. Should it?

2.4.3 PIXI.js

PixiJS is a web rendering library that will allow the user to create vibrant, interactive graphics, cross-platform applications. It has the features of creating cross-platform online application such as animation or games without having to deal with WebGL API directly. It can also handle the rendering in 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 animated graphics vibrant, interactive websites, applications it can be suitable for implementing animation for LPS program. Moreover, it is also cross-platform compatibility and easy to hook with the LPS interpreter LPS.js.

2.4.4 Express.js and REST API

Express.js is a minimal and flexible Node.js web application framework [20] which provides a robust set of REST API features for web and mobile applications. With a myriad of HTTP utility methods and middleware, creating a robust API is more natural and more light-weight compared to other back-end frameworks. It provided a range of built-in functions and structure for the user to develop and build the back-end side of the web application.
A RESTful API is an application program interface that uses HTTP requests to GET, POST, PUT, and DELETE data from the server [21]. RESTful API is needed when the user needs to perform the above actions in the LPS visuzliser system. It is also the bridge between the back-end program and the database.

2.4.5 MongoDB

MongoDB[22] is a document-oriented database. It has features of storing data in JSON-like documents, open-source, and cross-platform. Since a fully functional web application with REST API is going to be built. The database is essential for storing not only the user information in the LPS visualiser project but also the LPS program potentially. MongoDB is one of the most lightweight databases, and it supports cloud storage, so MongoDB is considered the most suitable database to use in LPS visualiser system.

3 Implementation

3.1 Overview

As I said before this needs to be split into smaller chapters. It is 30+ pages long at present, so at least 2 chapters worth. Apart from anything else it makes for too many sub sub sections.

This part is about the implementation of the LPS visualisation. This section consists 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 part is about the LPS program implement to fit various traffic situation. The fourth part is about the front side LPS program, including data parsing, data structure processing, and animation. The fifth part will be server-side programming, including using technology ExpressJS and some user authentication tools. Since the primary purpose of this project is to visualise the LPS program in traffic scenario lively, there are several key issues that need to be addressed before designing and implementing the program.

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

   To add 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 critical function in (Engine.js). It is also addressed by Sam Yong “lps.js does not support the construction of LPS programs by JS without explicitly writing LPS code 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 an asynchronous way.

2. What front-end tool is going to be used to achieve live animation? There is an 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 the crash of the program, unable to work with recursion etc.). The idea of using LPS studio is abandoned. After weeks of research, PIXI.js was found very suitable to animate the LPS program. This lead to another problem whether to use Pixi direct in HTML or use it in node.js by creating 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 too.

3. What information about the 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 following:

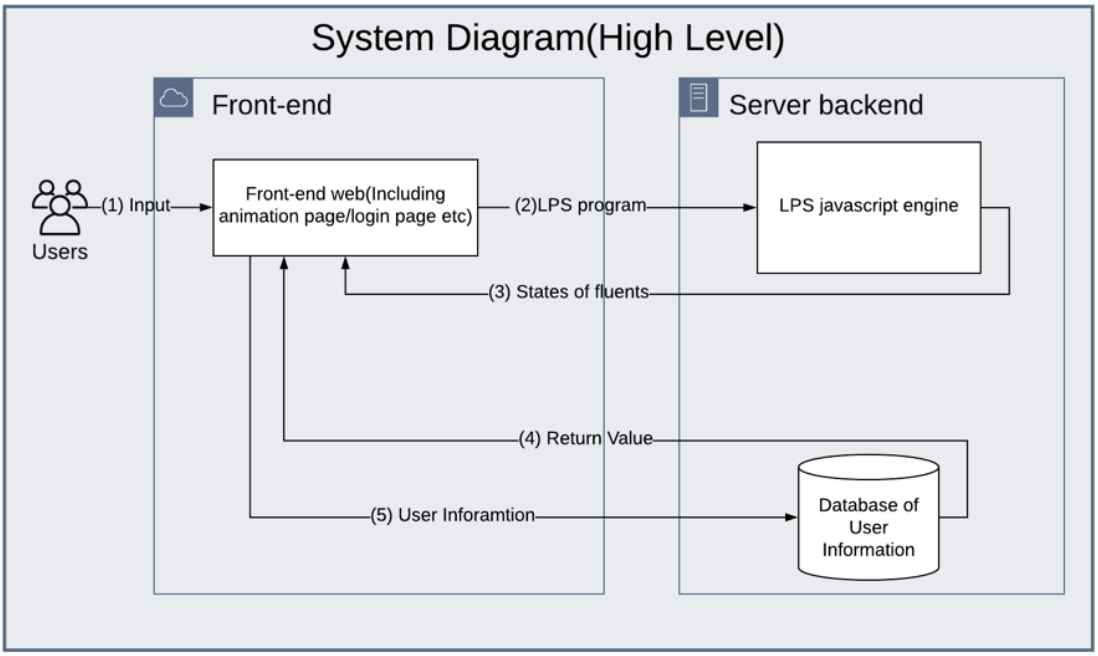
```
1 const LPS = require('lps');
2 LPS.loadString('...')
3   .then((engine) => {
4       engine.run();
5   });
```

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

capture the change of fluent in each running cycle. The section 3.5.1 and section 3.5.2 will illustrate detail in the following.

3.2 Structure design

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



This is quite high-level - can it be broken down into smaller program blocks?

Figure 10: 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.

3.3 Implementation using LPS studio and swish

Before implement the LPS visuliser, some other version of implementation are experimented in LPS studio and swish. The visualization example below is in 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)
```

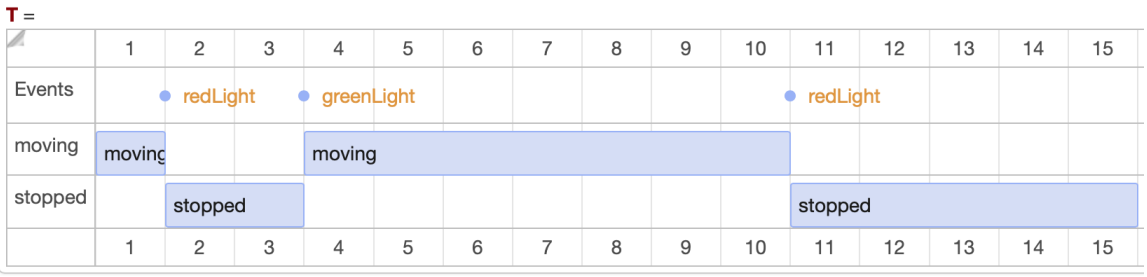


Figure 11: swish visualisation.

Fig. 11 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.

Swish has advantage of ability to show different event and fluent in a time line. However, when more and more object is introduced, the graph will be chaos and hard to read. The visualisation in swish also do not support real time animation. As the drawback mentioned above, swish visualisation was not going to be accepted in the implementation stage.

One car moving on straight line scenario in LPS studio

Figure 12: One car moving case.

Fig. 12 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).
terminates(greenLight, waitting).
initiates(arrival, stopped).
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.

```
lpsDefineObject(car, image, [
  position(150, 340),
  size(64, 64),
  isHidden(0),
  image(car),
  zIndex(100)
]).
moving(T1),loc(car,A,B) ->
  carMoveTo(car, A + 100, B) from T1 to T2,
  lpsAnimateMoveObject(car, 2, A + 100, B) from T2 to T3.
```

T-junction scenario in LPS studio
Two car moving towards the goal location in a T-junction case has been worked out as fig. 13. 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[15]: `https://github.com/lps-js/lps-studio`

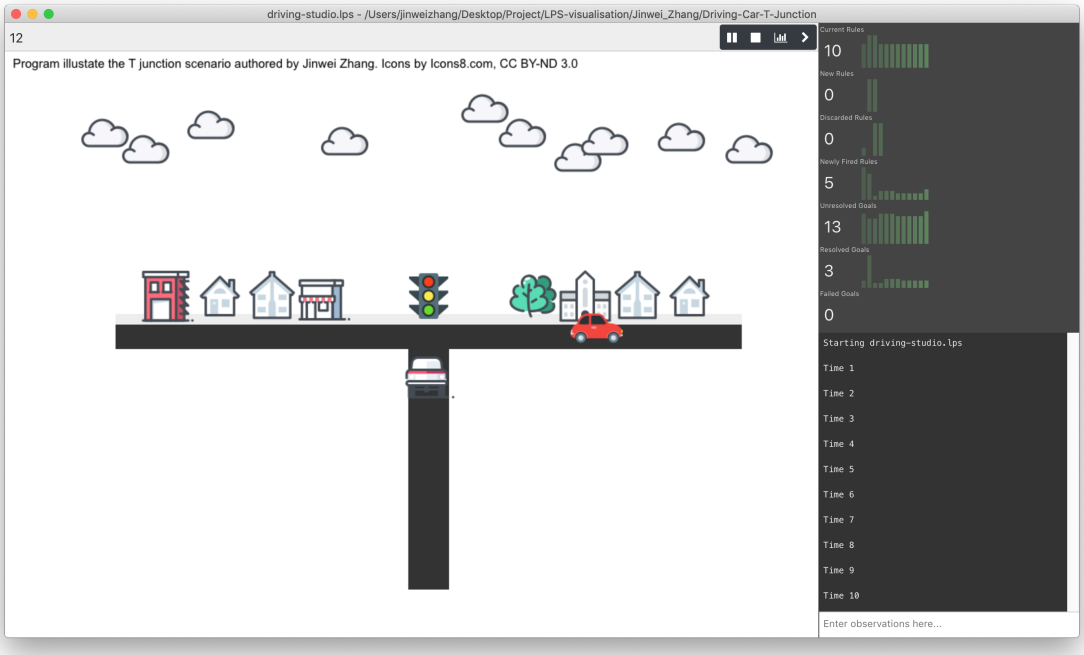


Figure 13: Two cars moving in a T junction.

Fig. 13 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.

The difficulty with the last few pages and this one is that it describes what you haven't done! At the very least this should be in a different chapter from what follows, which I presume is what you have done.

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.

Is the above part of the caption for Figure 13? Put caption in italics so it is clear.

```
% 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)).
```

```
38
39 initiates(redLight, waitting(Obj)).
40 terminates(redLight, moving(Obj)) .
41 initiates(greenLight, moving(Obj)).
42 terminates(greenLight, waitting(Obj)) .
```

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.

```
1 lpsDefineObject(car1, image, [position(150, 340),size(64, 64),
     isHidden(0),image(car1),zIndex(100)]).
2 lpsDefineObject(car2, image, [position(520, 650),size(64, 64),
     isHidden(0),image(car2),zIndex(100)]).
3
4 %car moving sepraate the first car and the second car
5 goal_location(car1, A, B), car_location(car1,X,Y,T), X@<A, moving(
     car1,T) ->
6    NextX = X + 64,
7    carMoveTo(car1, NextX,Y) from T to _,
8    lpsAnimateMoveObject(car1, 2, NextX, Y) from T to _.
```

3.4 LPS program

This is the first place you have mentioned how LPS is written in this project. Please first give an overview of the various features required. i.e. list of fluents observations, actions etc. and what syntax is used.

3.4.1 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:

I don't see upper-case text?

```
1 fluents([
2    stopped(VehicleName),
3    velocity(VehicleName, Speed),
4    moving(VehicleName),
5    coordinate(X, Y),
6    location(VehicleName, coordinate(X, Y), Direction),
7    trafficLight(coordinate(X, Y), Working_status, Color,
        FacingDirection),
8    street(StreetName, coordinate(X, Y), Width, Height, Number_of_lane
        ),
9    goal(VehicleName,coordinate(X, Y))
10 ]).
```

```
1 % we assume the destination is reachable
2 maxTime(Number).
3 cycleInterval(Number).
4
5 fluents([
6    Set of fluent predicate defined
7 ]).
8
9 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 the map is drawn on a canvas, say the size in pixels, etc. You can even show a canvas with the above street and vehicle shown in particular. I suggested all this previously.

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).

3.4.2 Car moving straight

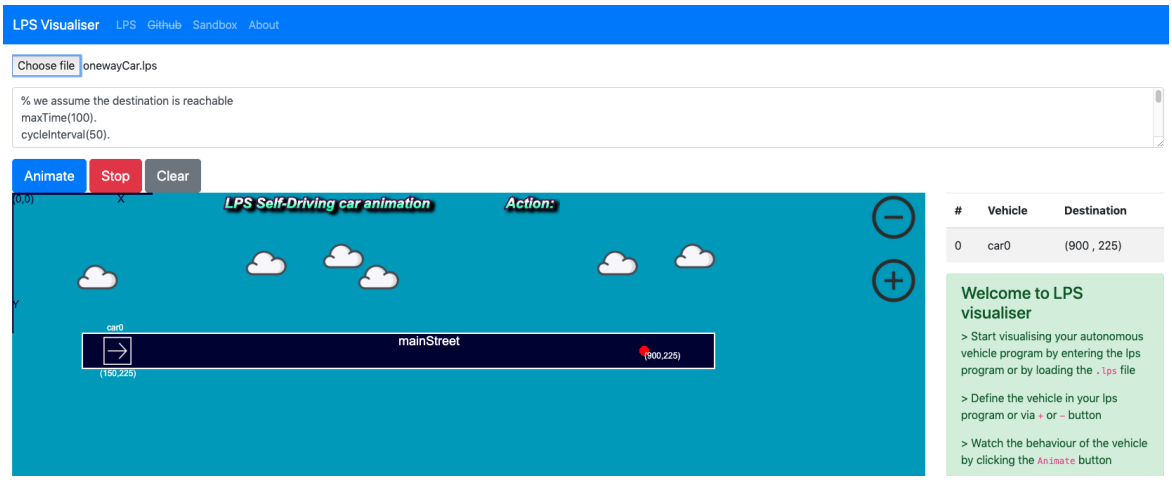


Figure 14: show a car can move straight towards the goal (red point)

This the the straight moving LPS program and it is able to be visualised on web.

```
fluents([
  stopped(VehicleName),
  velocity(VehicleName, Speed),
  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)
]).
initially([
  moving(car0),
  location(car0, coordinate(150, 225), eastward),
  goal(car0,coordinate(900, 225)),
  velocity(car0, 5)
]).
street(mainStreet , coordinate(100, 200), 900, 50, 1).
```

The code snippet above defined a set of fluetns and actions. Then in the initially part, the initial fluetn are defined. An example of a fact predicate – street is also defined as above.

```

goal(Vehicle,coordinate(A, B)) from _ to T,
location(Vehicle, coordinate(X, Y), Direction), A==X, B==Y,moving(
   Vehicle) at T ->
```

```
4    % testPrint(Vehicle+ ' we have arrived'),
5    arrive(Vehicle) at T.
6
7
8  goal(Vehicle,coordinate(A, B)) from _ to T,
9  location(Vehicle, coordinate(X, Y), Direction), moving(Vehicle) at T
       ->
10   % need to find the right direction here
11   % driving forward
12     drive(Vehicle) from T to _.
```

I really think that you need to explain more the syntax of programs. For instance in Lines 8 and 12 what do the anonymous variables indicate?

The main reactive rule is defined above. The reactive rule checks if the goal has been reached if not then keep driving towards the current direction.

Link the description to the line numbers in the code snippet. In fact, each of these code snippets should be labelled, say as Listing 1, etc.

The first reactive is defining if the vehicle is at goal (A,B) then this vehicle will perform the arrive action.
The second reactive rule defined as : If the vehicle has moving fluetn then it will perform "drive" action.

```
1  drive(Vehicle) from T to T1 <-
2    location(Vehicle, coordinate(X, Y), Direction),
3    Direction == northward,
4    NewY = Y - 10,
5    NextPlace = coordinate(X, NewY),
6    step(Vehicle, NextPlace) from T1 to T2.
```

You need to explain that each action moves a vehicle a few pixels in each time step. I don't follow what T2 is in the above?

One example of the decomposition of action is defined above, saying if the vehicle is facing north the vehicle the Y axis will reduce 10 pixcel then perfor the step action. There are four direction is the one-way car driving case so the decompositon of 'drive' action have 4 in total.

Anyone reading this who is unfamiliar with LPS will be completely lost, I think.

```
1  updates(step(Vehicle, NextPlace), location(Vehicle, OldPlace,
       Direction), location(Vehicle, NextPlace, Direction)).
2
3  updates(turn(Vehicle, NewHeading) , location(Vehicle, Place,
       OldHeading), location(Vehicle, Place, NewHeading)).
4
5  terminates(arrive(Vehicle), moving(Vehicle)).
6  initiates(arrive(Vehicle), stopped(Vehicle)).
```

When step action is triggered which will terminate the old location and initialize a new location. When arrive action is triggered, It will terminate the moving state of the vehicle and initiates it a stopped state such that this vehicle can not move anymore.

3.4.3 T junction with both narrow lane

I would make a new chapter here and also before 3.5.

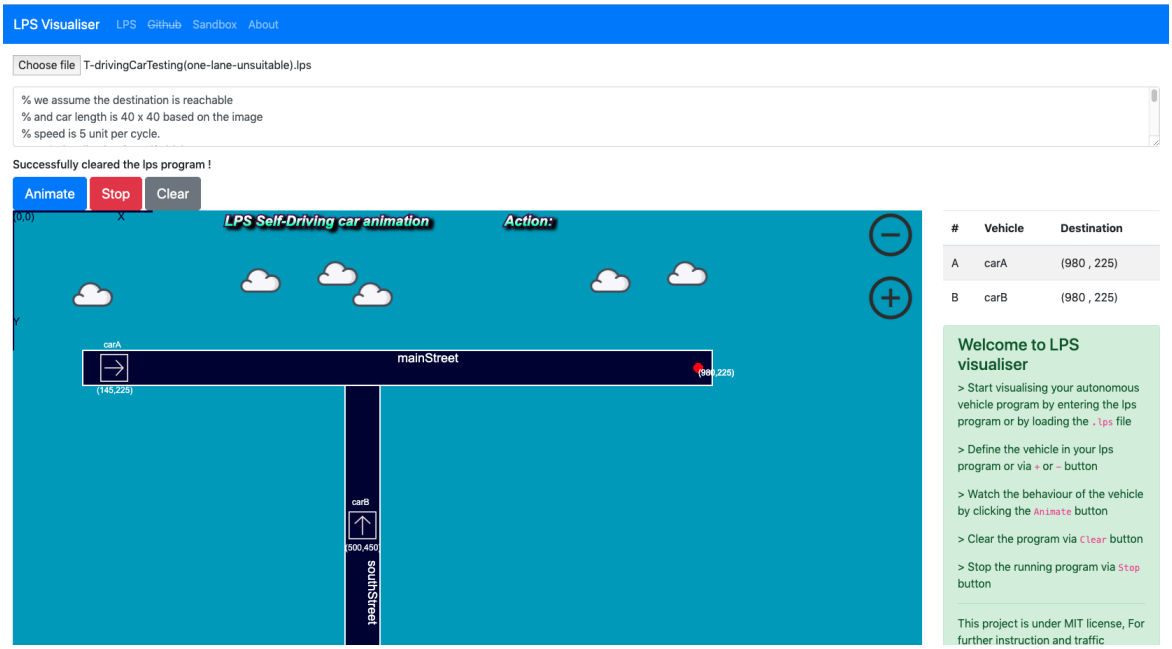


Figure 15: In the graph the red dot represent the goal location of the car. In this particular example both carA and carB has the same goal of (980 225). However when both car reached the T junction carB should give way to the carA because the car on the main road has higher priority in this case.

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

This doesn't make clear what is happening and how it relates to the picture above.

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

3.4.4 T junction with one narrow lane and a double lane

1. Two car:

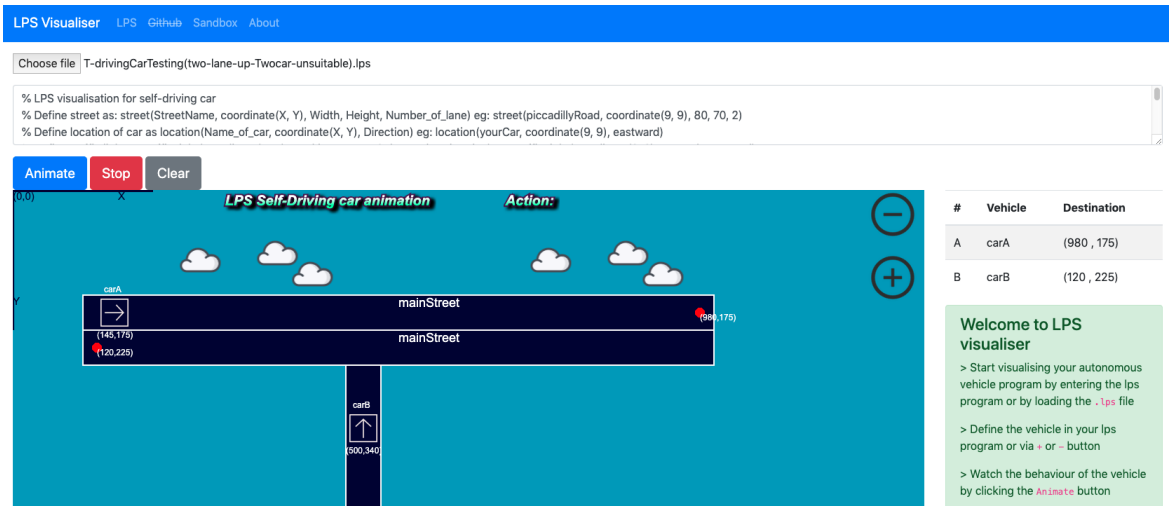


Figure 16: This is similar to the previous T junction with both narrow lane however a new road and junction need to be defined in this case.

2. Three cars:

Figure 17: 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

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

Is it not possible to explain with reference to the line numbers?

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. 18:

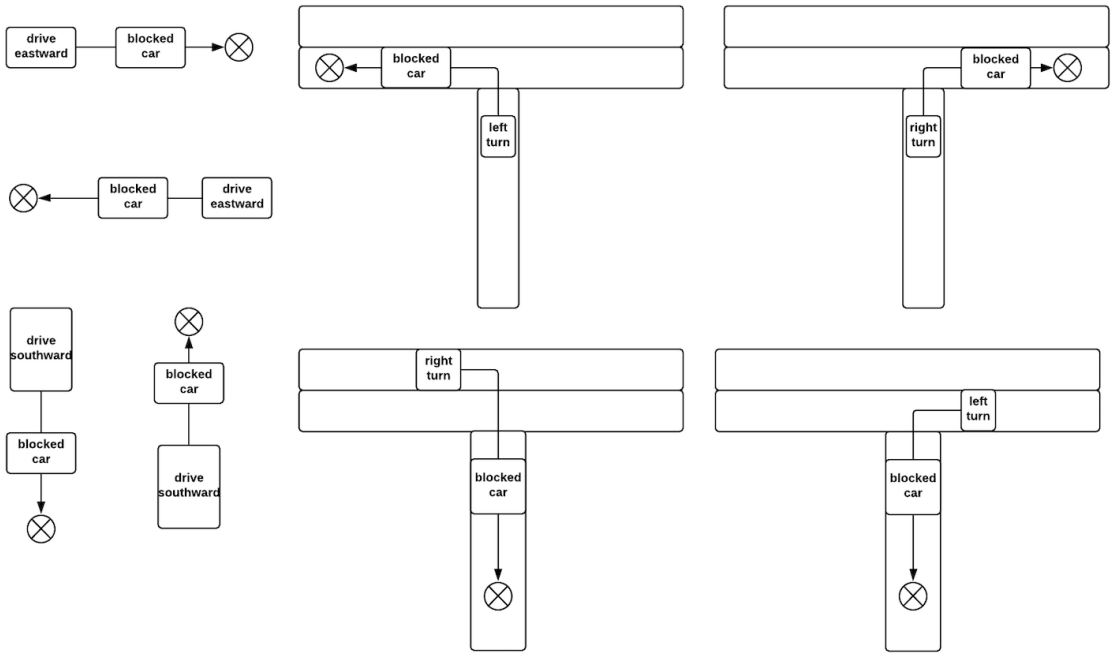


Figure 18

```
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.
```

Again, relate to the line numbers of the code.

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

```
1 % both car have clear routes but car on the narrow road can not
    step
2 % (car on main street has higher priority)
3 <- step(Vehicle, NextPlace),
4   not onMainRoad(Vehicle) at T,
5   collisionPossible(Vehicle, Vehicle2) at T,
6   Vehicle != Vehicle2,
7   onMainRoad(Vehicle2) at T,
8   not blockedRoute(Vehicle,Vehicle2) at T,
9   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.

3.4.5 T junction with two double lanes

Figure 19: 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.

```
1 trafficLight(coordinate(X, Y), Color, FacingDirection,Street) at T->
2   opposite(Color,Color2),
3   Reminder = mod(T,40),
4   Reminder == 0,
5   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).

```
1 % updating the traffic light
2 updates(changeTrafficLight(Place,Color) , trafficLight(PlaceIns,
     OldColor, _, _), trafficLight(PlaceIns, Color, _, _))<-
3   % testPrint('traffic light as been updates'),
4   Place == PlaceIns.
```

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

```
1 <- step(Vehicle, NextPlace),
2   location(Vehicle, coordinate(X, Y), Direction1),
3   trafficLight(coordinate(A, B), red, Direction2, Street),
4   X==A,
5   Y==B.          Simple equality atoms like this can be put on one line to compact the code a little more.
```

Restriction of step action need to be added since vehicle can not go forward when

Does this mean the car is not stopped, but still moving, except that it is not allowed to move?

the traffic light is red

3.4.6 Cross junction with traffic light

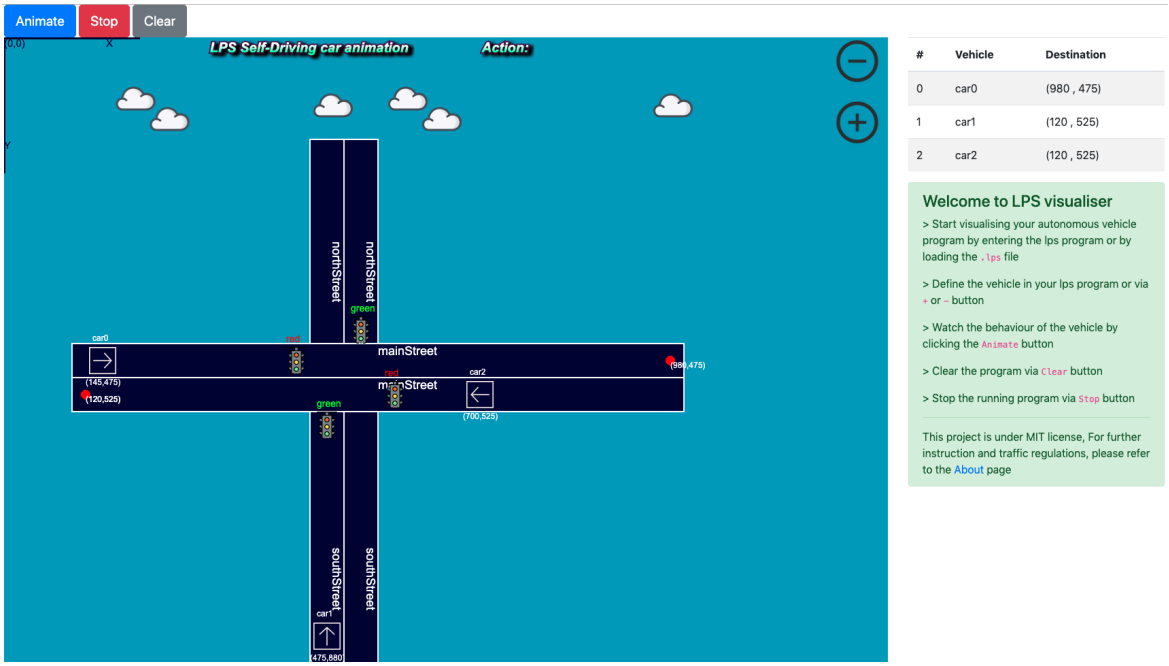


Figure 20: In the case of the cross junction there are still 4 junction block in total so the reactive rule and the restriction of the step action is the same as the previous two double lane T junction road. No more reactive rule is introduced here.

3.4.7 Traffic with restart feature

Explain why this is achieved by restart has been designed

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.

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

For the traffic light with restart feature

```
1 % if the car is stopped and it has a start location.
2 stopped(Vehicle) at T ,startLocation(Vehicle, _, _)->
3   restart(Vehicle) at T.
4
5 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 from the starting point. Hence a new action is also need to be written as follow:

Unclear - a car stopped at a red light shouldn't restart from start location should it?

3.4.8 Other Traffic road

Other kinds of Road layouts

To introduce more complex traffic road, The road interface need to re-implemented. Moreover, the direction will be replaced by vectors in 2D coordinate system.

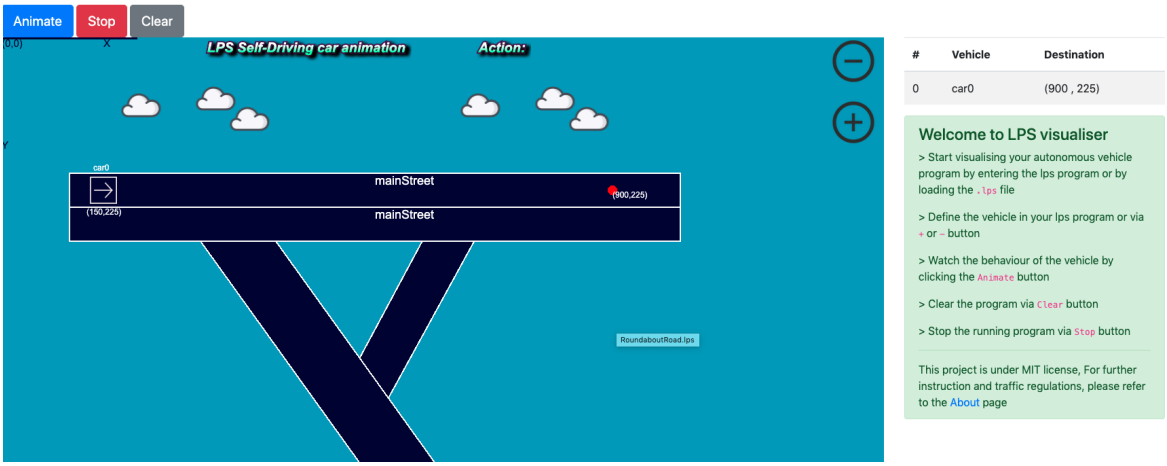


Figure 21

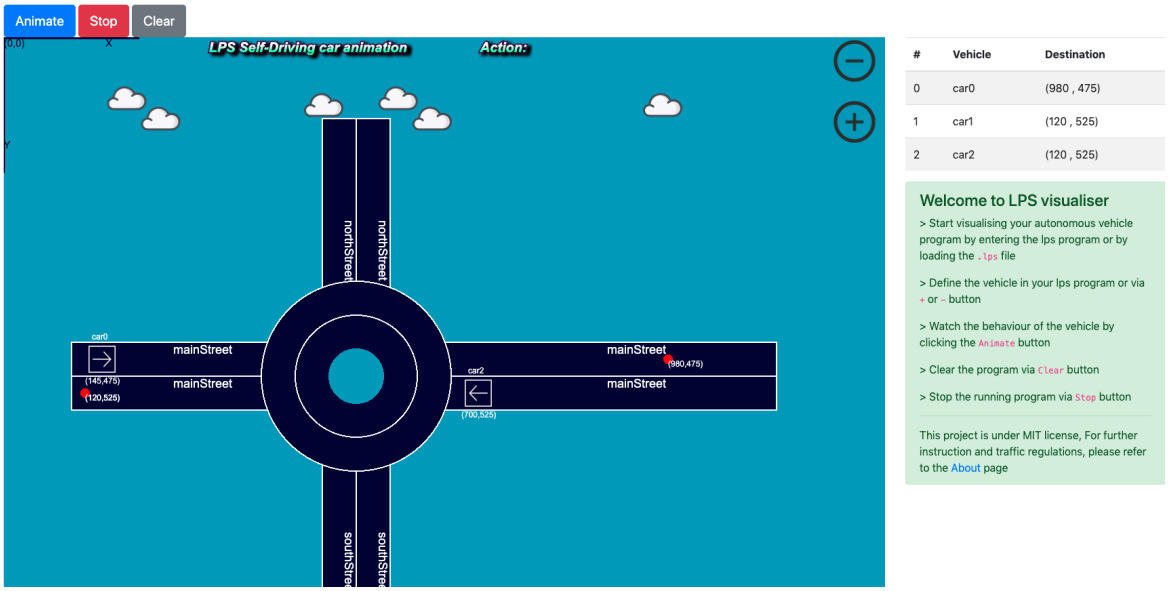


Figure 22

Fig. 22 and Fig. 21 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.

To animate differnt direction in differnt time cycle, a vector rotation action need to be formalised. In a two-dimensional coordinate system, the rotation formula of a position vector can be derived from the geometric meaning of the trigonometric

function.
For example, the figure below shows the situation where the position vector R rotates counterclockwise by the angle B. In the left picture, we have a relationship:

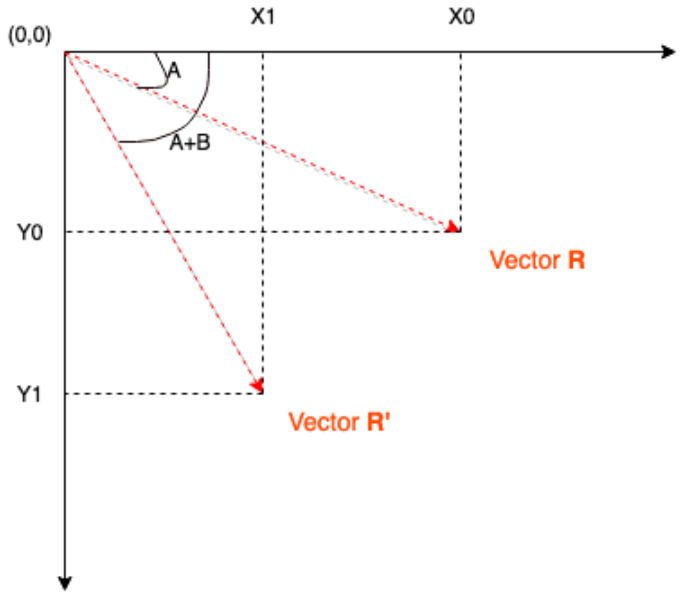


Figure 23: The above figure shows the situation where the position of vector rotates clockwise by the angle B. The coordination system is the same as the PIXI.js canvas. The vector can represent the direction of the vehicle meaning vehicle has turned from direction (X0,Y0) to (X1,Y1). It has turned B degree into southward direction.

Based on Fig. 23 we have following formula:

$$X_0 = |\boldsymbol{R}|\cos(A) \quad \Rightarrow \quad \cos(A) = \frac{X_0}{|\boldsymbol{R}|} \tag{2}$$

$$Y_0 = |\boldsymbol{R}|\sin(A) \quad \Rightarrow \quad \sin(A) = \frac{Y_0}{|\boldsymbol{R}|} \tag{3}$$

$$X_1 = |\boldsymbol{R'}|\cos(A+B) = |\boldsymbol{R'}| * (\cos(A)\cos B - \sin A \sin B) \tag{4}$$

$$Y_1 = |\boldsymbol{R'}|\sin(A+B) = |\boldsymbol{R'}| * (\sin(A)\cos B + \cos A \sin B) \tag{5}$$

Now substitute (1)(2) into the above expression to get:

$$X_1 = |\boldsymbol{R'}| * (\frac{X_0}{|\boldsymbol{R}|}\cos B - \frac{Y_0}{|\boldsymbol{R}|}\sin B) = X_0\cos B - Y_0 \sin B \tag{6}$$

$$Y_1 = |\boldsymbol{R'}| * (\frac{Y_0}{|\boldsymbol{R}|}\cos B + \frac{X_0}{|\boldsymbol{R}|}\sin B) = Y_0\cos B + X_0 \sin B \tag{7}$$

In this way we get the clockwise rotation formula of the vector around the dot in the two-dimensional coordinates. The matrix form of rotation formula is written below.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{8}$$

using formula (6) and (7) the direction can be change to any direction.
By defining these fact and fluent at the beginning.

```
1 roundabout(StreetName, coordinate(X, Y),Radius1,Radius2,Radius3), %
     Fact
2 dir(A,B), % Fluent
3 location(VehicleName, coordinate(X, Y), dir(A,B)), % Fluent
```

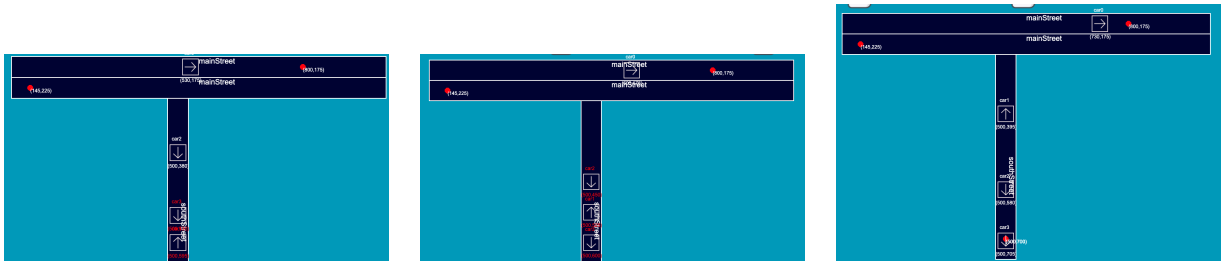
re-define the change direction event.

```
1  % you only turn when you are in the middle of the juction.
2  change_direction(Vehicle) at T<-
3   location(Vehicle, _, dir(A,B)),
4   goal(Vehicle,coordinate(K, P)),
5   Theta = pi/36, % Theta is the angle turned each time define/
       calculated by user
6   X = A*cos(Theta) - B*sin(Theta),
7   Y = B*cos(Theta) + A*sin(Theta),
8   turn(Vehicle,dir(X,Y)) at T.
```

The all angle direction rotation can be achieve through the lps code above. The angle that turned each time cycle depends on the user and the shape of the road.

This is quite nice

3.4.9 Passing through



((a)) before **((b))** passing **((c))** end

Figure 24: 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.

Should say something about what this simulates in real life!

3.4.10 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.

Introduce the topic of overtaking -If overtaking is to be simulated, then this necessitates that the overtaking car must have a greater speed than the car being overtaken. In turn, this requires that the speeds of vehicles need to be maintained as a fluent.

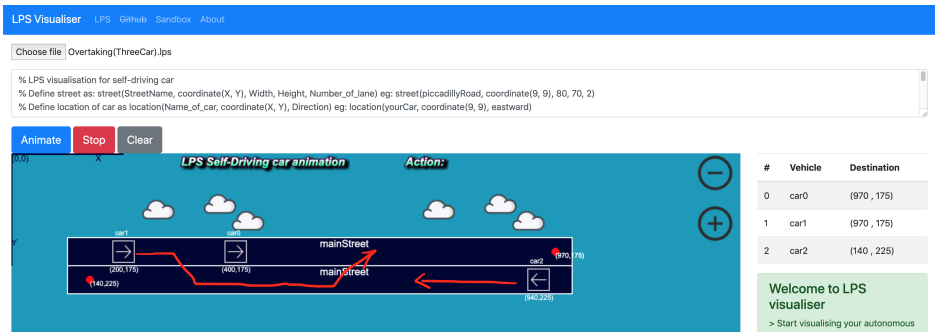


Figure 25: as describe in the background (section?) the car need to check if there another within its overtaking route if the overtaking route is clear the overtaking can perfume overtaking. If not car will follow in the behind of the car. As the example above, car1 will perform the overtaking because car2 will not collide with car1 during the overtaking action.

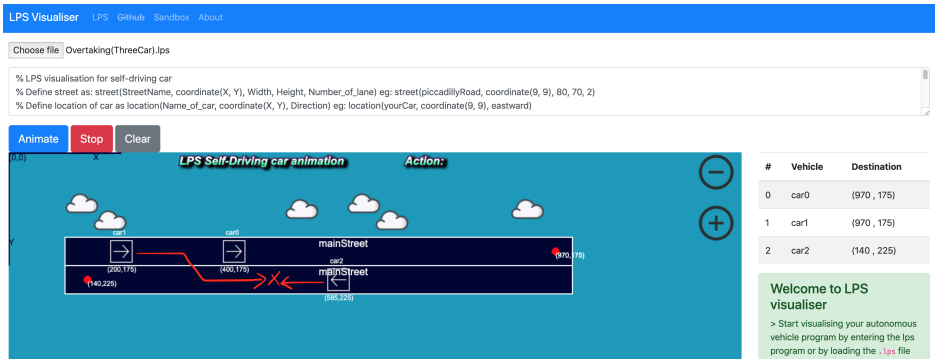


Figure 26: In the example above car1 will not perform the overtaking

```
collisionPossible(Vehicle, Vehicle2) from T1 to T3,
  Vehicle != Vehicle2,
  velocity(Vehicle, S1),
  velocity(Vehicle2, S2),
  S1 > S2,
  blockedRoute(Vehicle,Vehicle2) at T1 ->
  overtaking(Vehicle,Vehicle2,T1,T3).

overtaking(Vehicle,Vehicle2,T1,T3) <-
  location(Vehicle, coordinate(X2, Y2), _) at T1,
  NewY = Y2+50,
  NewLocation = coordinate(X2, NewY),
  aheadClear(Vehicle,Vehicle2,Vehicle3),
  not goalTooNear(Vehicle),
  step(Vehicle, NewLocation,T2, T3).

navigateBack(Vehicle,T1,T3)<-
  location(Vehicle, coordinate(X2, Y2), _) at T1,
  NewY = Y2-50,
  NewLocation = coordinate(X2, NewY),
  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.

I am sorry to keep on mentioning it, but definitely this is a place to start a new chapter.
It will save so many sub sub sections too.

3.5 Front-end logic design and animation

This chapter will

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.

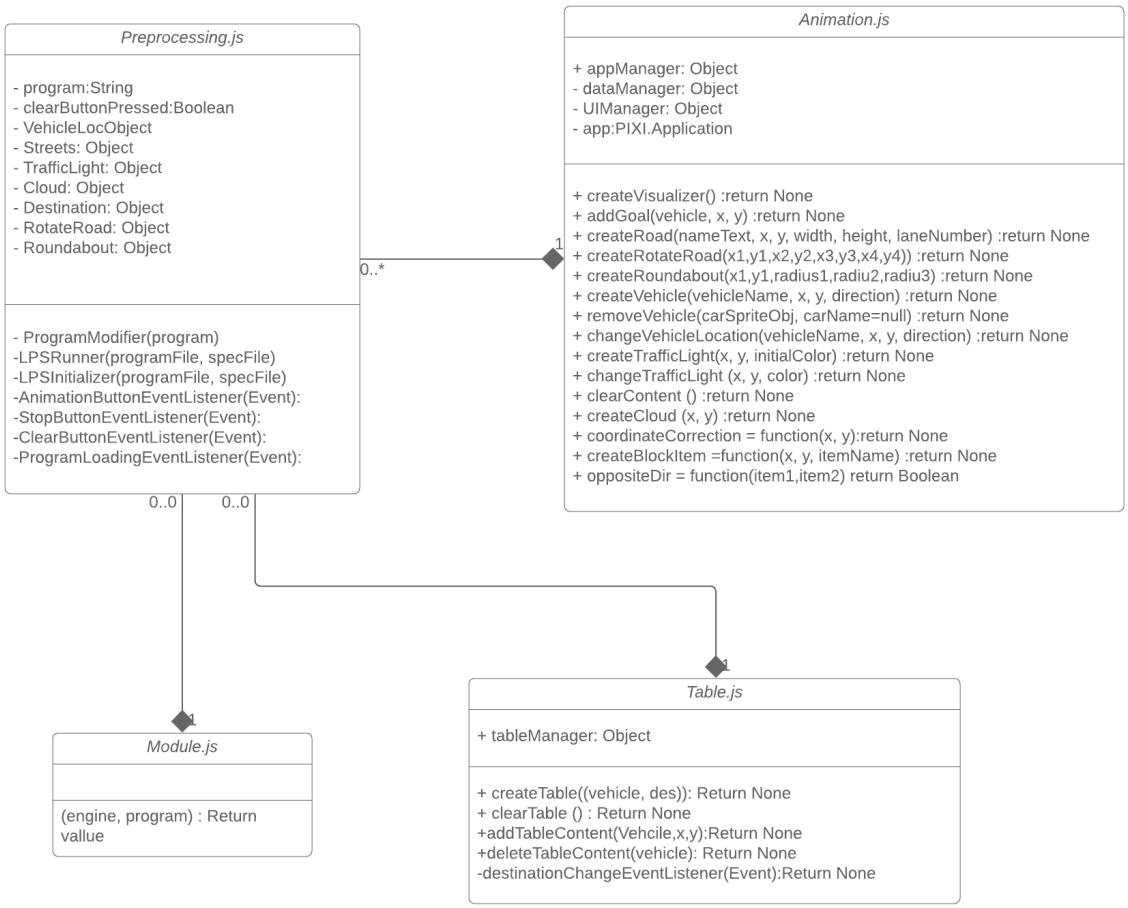


Figure 27: The class diagram of the front-end side is structured as above. The detail explained will be separated described below.

explanation

3.5.1 Animation.js

3.5.1.1 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.

Introduction doesn't need a header!

3.5.1.2 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.

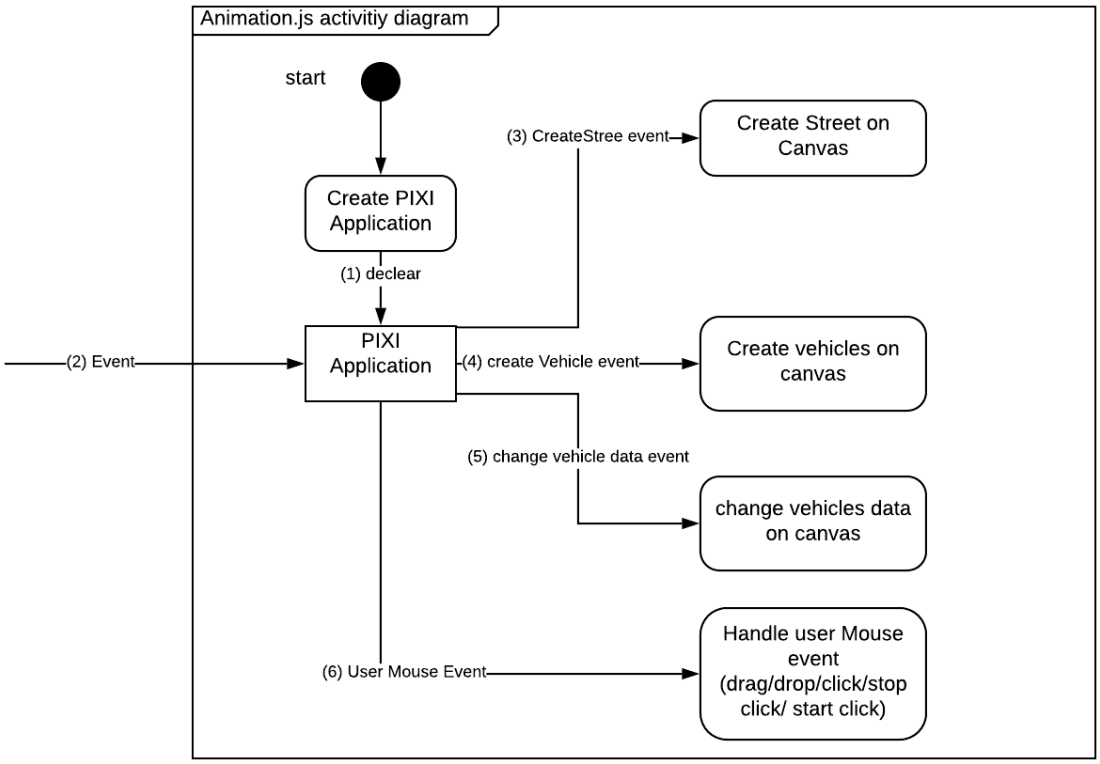


Figure 28

Even if these points are numbered, they would be better put into a narrative linking them together. "The various events in Figure 28 can b characterised as follows: (i) Creating the PIXI application; (ii) classifying an event??; (iii) etc.

(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. Do you mean street creation is handled when a street atom is input?

(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.

3.5.1.3 Class fields ?

Figure 29

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

3.5.1.4 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. Parameter meanings?

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.

This might look better if you italicise the function name. That way you can avoid the space between each case, which is annoying to read.

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).

3.5.2 Preprocessing.js

3.5.2.1 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.

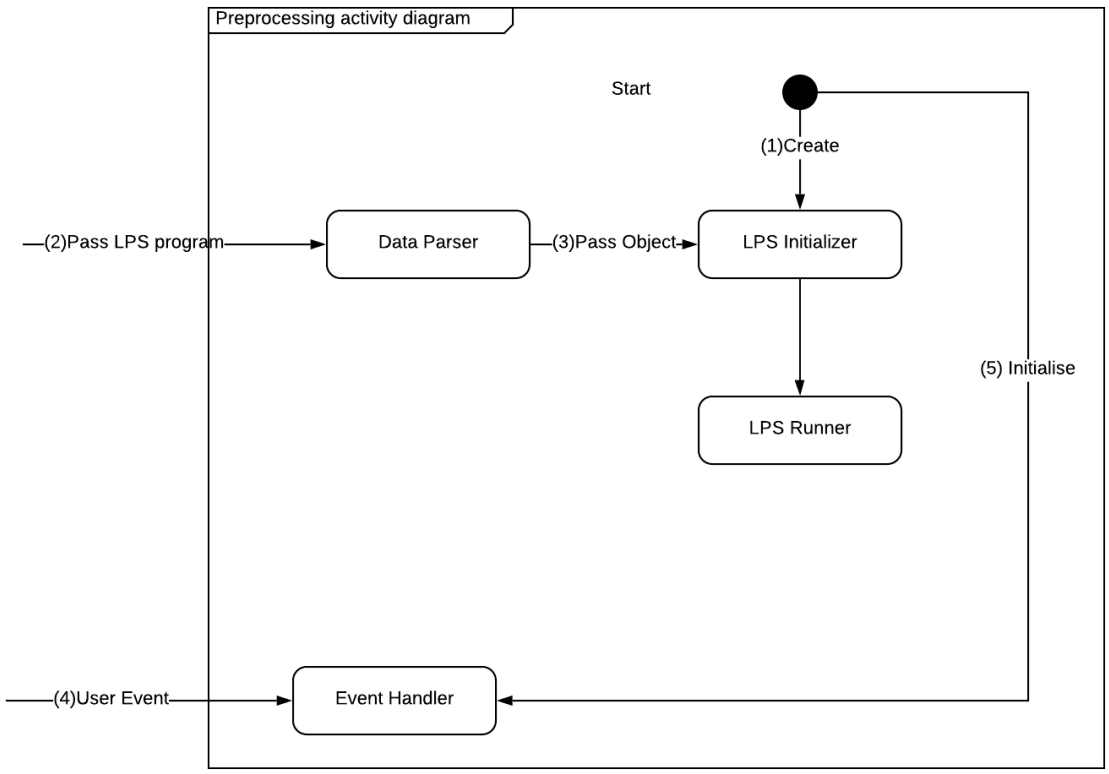


Figure 30

3.5.2.2 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.

3.5.2.3 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 [23]

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

Probably doesn't need its own sub sub section

3.5.2.4 Functionality

This part will be the functionality description:

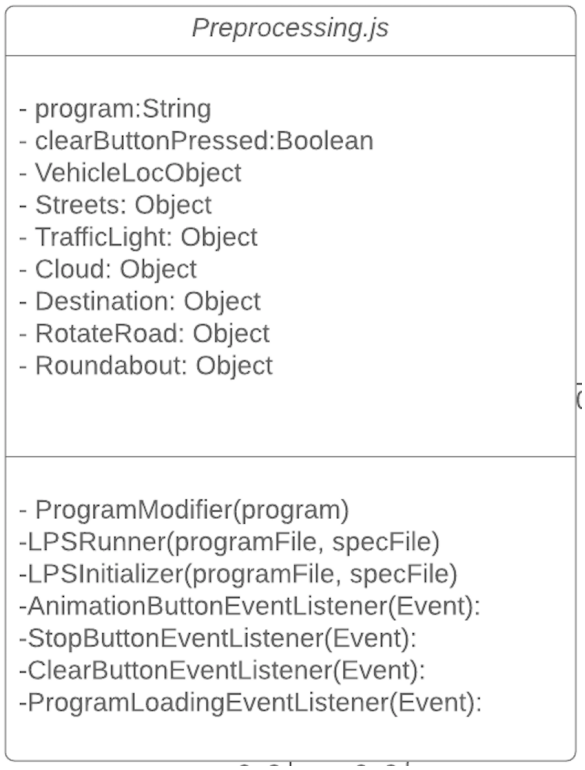


Figure 31: class diagram

ProgramModifier(program) capture the change by the user before starting the LPS simulation. User might have deleted/added/modified the vehicle or the property of the vehicle. ProgramModifier need to capture those change and modify back to the LPS program.

LPSRunner(programFile, specFile) take the program string as input and run the LPS program asynchronously, the key inforamtion such as the location of the vehicle need to be parsed into animation.js class to animation on the web simultaneously.

LPSInitializer(programFile, specFile) take the program string as input. It will render a LPS program when the program is on cycle 1. This will allow user to do some modification such as vehcile deletion, goal changing.

AnimationButtonEventListener(Event): This event listener will listen for user click event using DOM strucutre. When user click on the animation button. The LPS pro-

gram should be updated and run.

StopButtonEventListener(Event):This allow user to stop the runnig LPS program at any time. When the user click the animation button, the LPS program will start again.

ClearButtonEventListener(Event) allow user to clear all the web cache, the PIXI canvas, the tabe content as well as the LPS program text input area.

ProgramLoadingEventListener(Event) allow user to click on the load program button to load a new program. If there is already a program exist, the old program will be overwrite and replaced by redndering a new PIXI canvas.

3.5.3 Table.js

3.5.3.1 Introduction Table.js manage any of the event that need modification of the table. For example, when LPS program is loaded, the table need to be created. When user clear the content by clickging the clear button, the table data need to be cleared etc.

3.5.3.2 Functionality The functionalities will be shown as follow:

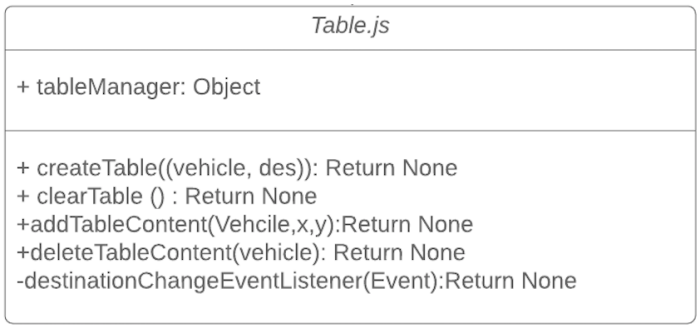


Figure 32: The class diagram of tableJS

The brief description of the functionality:

CreateTable(vehicle,des) takes the vehicle name and the destination array as input. It will create a dynamic and editable table on the main web page.

addTableContent(Vehcile,x,y) takes the vehicle name,x and y coordinate as input. The purpose of this method is to append the new destination onto the table array.

deleteTableContent(Vehcle): As we describe in animation.js there is a vehicle deletion function, when the vehicle is deleted the corresponding goal data on the table need also be deleted.

destinationChangeEventListener(Event) will listen for event of user changing the destination of any of the vehicle, the corresponding data in Animation.js also need to be changed.

3.5.4 Demonstration Website design

This seems to be a sequence of pictures. At least say that the sequence captures the screens seen by a new user when they run the application (by typing ....)

The following subsection will demonstrate the layout of the application, including the login page, registration page, the mobile version page and the about page. The technology involved will also be introduced accordingly.

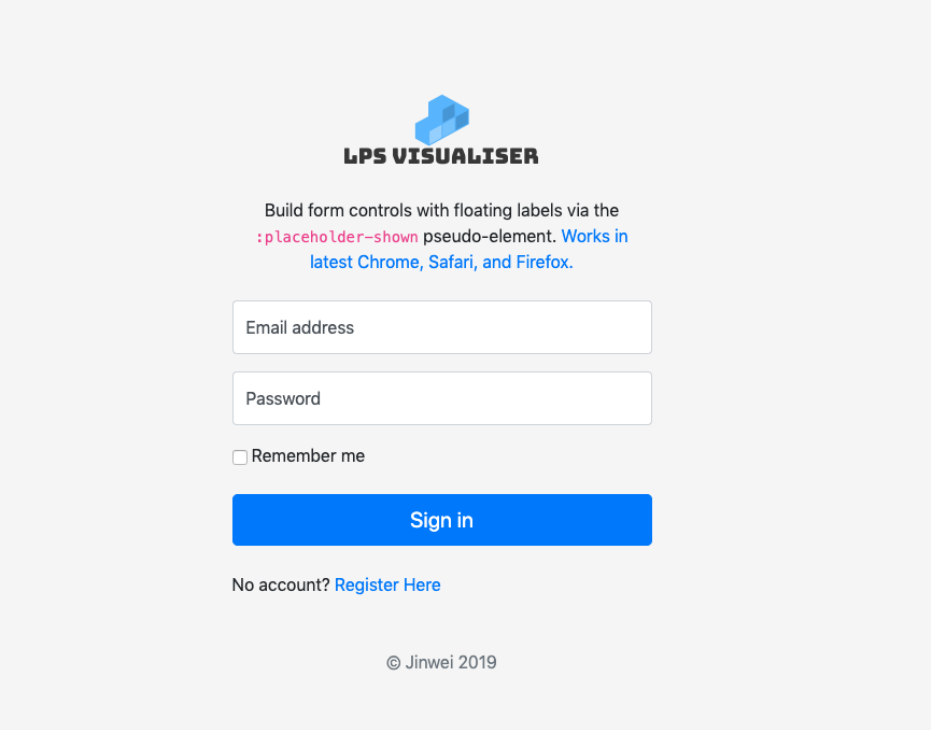


Figure 33: The login page, The login page used the twitter Bootstrap login template, with new design logo.

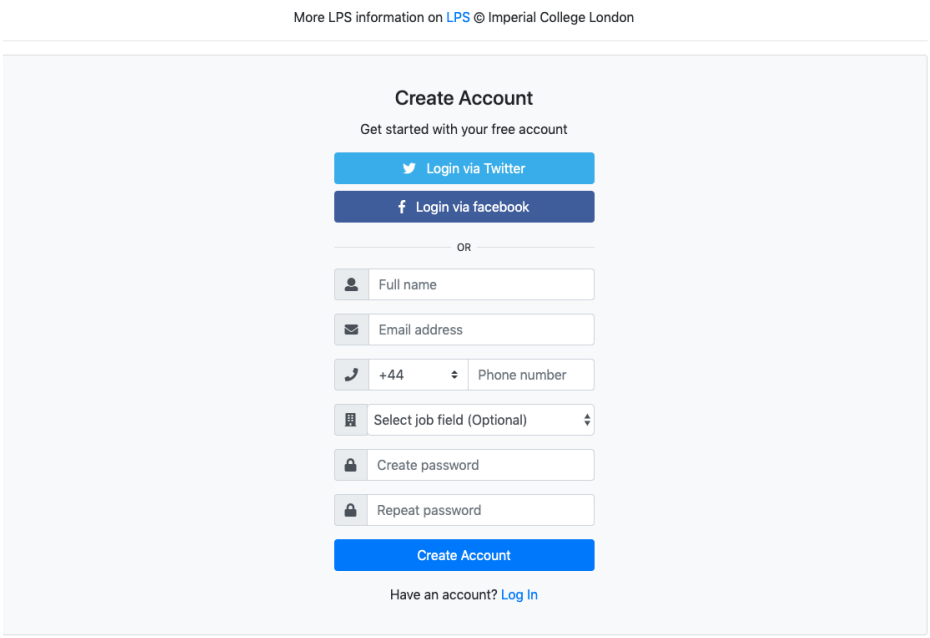


Figure 34: the registration page. The registration page maintain the same theme with the login page and the following main page with a Jekyll generator. The css comes from Frontawsome css as well as the Boostrap css.

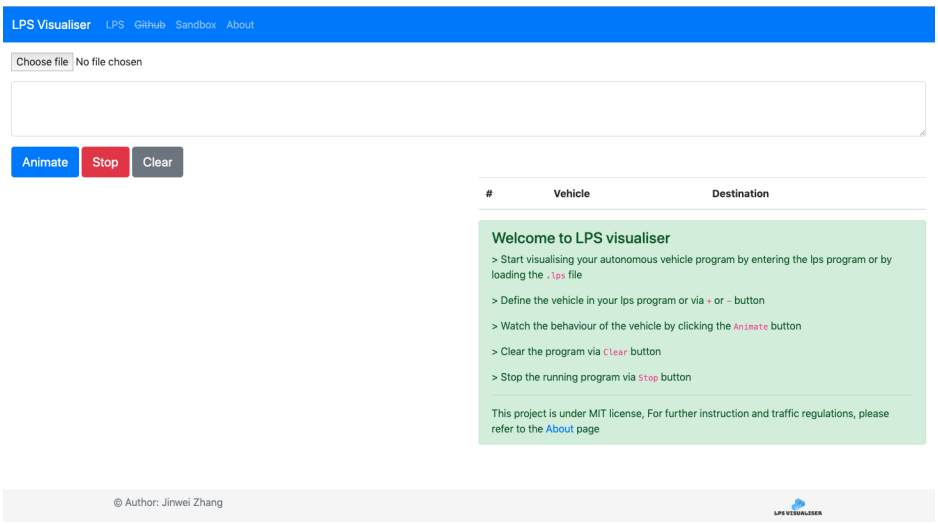


Figure 35: The main page. Button, header and the green alert box are using built in Boostrap class.

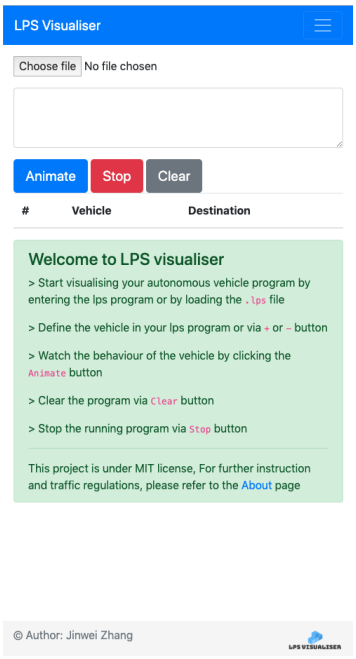


Figure 36: Mobile version. All the page including the main page is mobile friendly. Using the fluid container of Boostrap, the content can be layout into the mobile format.

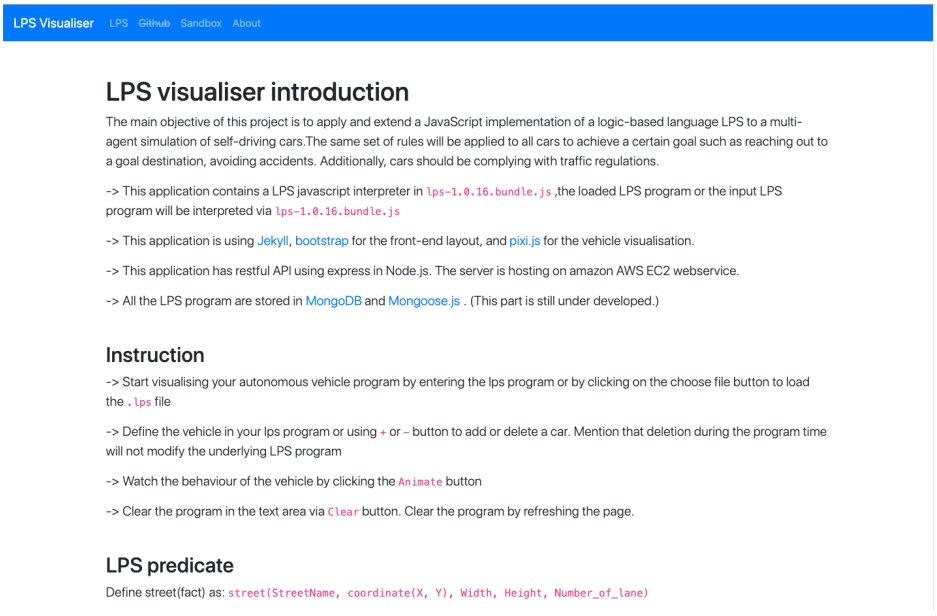


Figure 37: About us page. This page give a brief introduction of how to run the program and how to define LPS predicate etc

3.6 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.

3.6.1 Back-end hosting

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

3.6.1.2 Structure The graph below will demonstrate the structure of the back-end side of LPS visualiser.

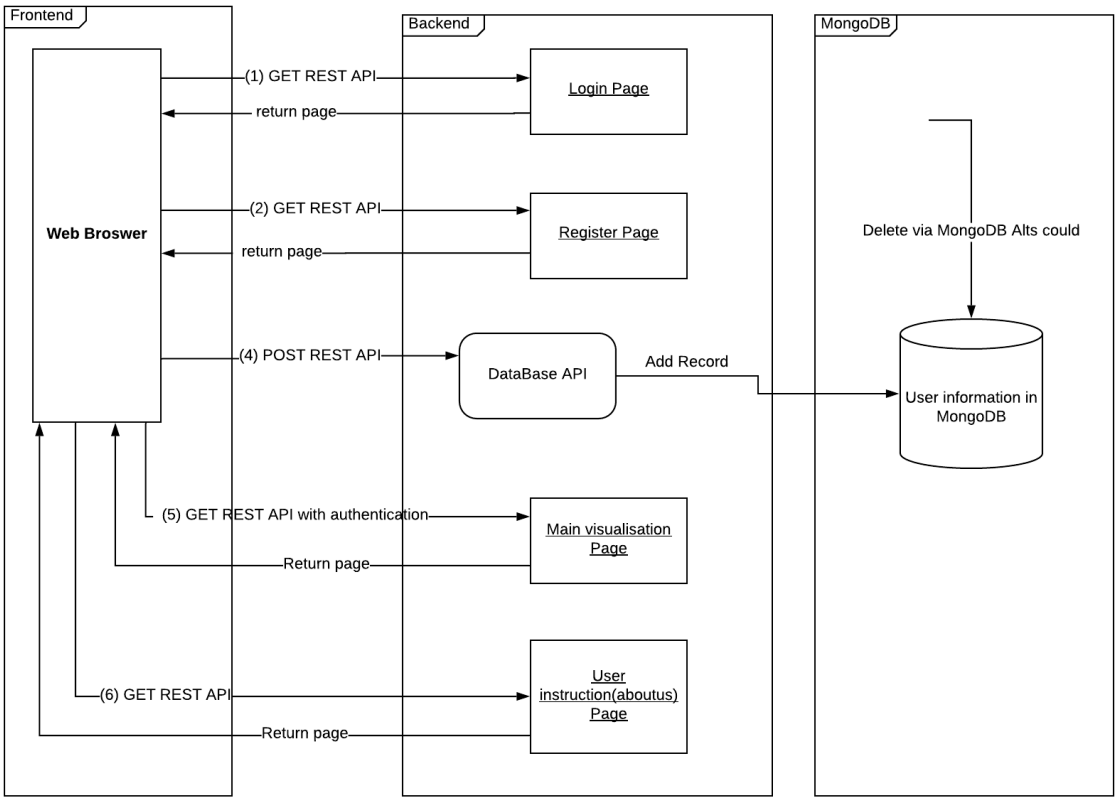


Figure 38: (1) is the event of user request the page by performing GET REST API action. After that a login page will return to the user's browser. This step is same for (2) and (6). (4) is for posting data in registration, user will perform POST REST API action to the database API then the data will be append onto the MongoDB cloud database. (5)After registration, the main page can be requested using authorized username and password. Notice that the LPS.js interpreter and the LPS visualiser will all be returned as JS file

3.6.1.3 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.

```
1 const UserSchema = new mongoose.Schema({
2     name: {
```

```
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    phone: {
        type: Number,
        required: true
    },
    job: {
        type: String,
        required: false
    },
    password: {
        type: String,
        required: true
    },
    date: {
        type: Date,
        default: Date.now
    }
});
```

3.6.1.4 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.

3.6.1.5 AWS EC2 server ?

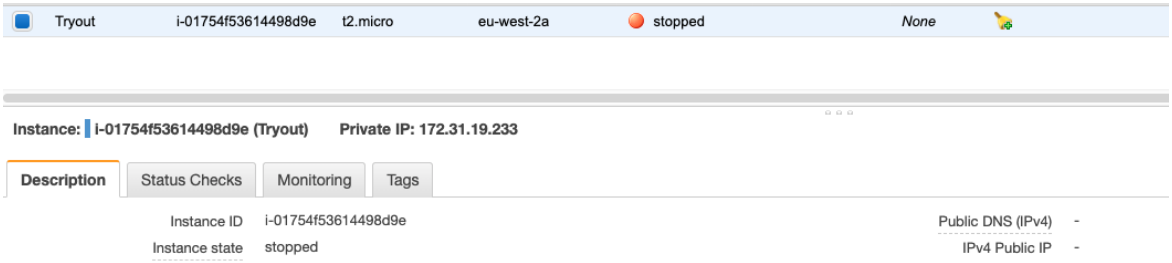


Figure 39: The server is EC2 instance, the server will provide a public IP(IPV4) once hooked with the port number, the website is ready to go online.

4 Evaluation

4.1 Usability

The about us page introduced all the predicate defining standard the, and there is also a tutorial box on the main page. The user who has no programming experience is easy to follow the educational level tutorial. Then the user can program in LPS and visualise the code after going through the tutorial provided. The developer is also able to view and develop the project with the well-documented system.
These features allow us to claim that the application is easy to use.

4.2 Efficiency

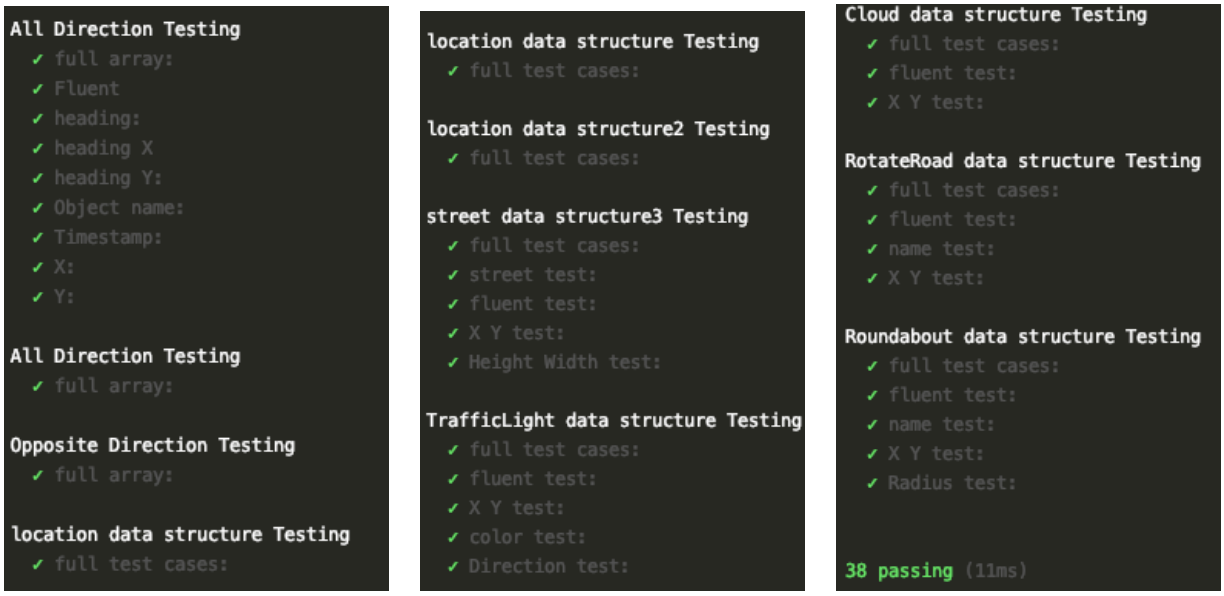
The program is hosted on a server and running in an average of 0.1 seconds of request time. The asynchronous functions have been used instead of synchronous by the promised feature in order to avoid unnecessary waiting time.

4.3 Re-usability

Not a sentence
As we have described the problem about the previous visualisation tool LPS.studio and the LPS web version. This project is easy to extend, unlike the interpreter engine LPS.js. The code for LPS visualiser is well-documented. There are no lengthy functions and useless variable and unnecessary function call-back defined in LPS visualiser project. LPS visualisation have kept the code simple to use and well structured. Testing should be formed to testing some critical boundary cases moreover the unit testing should be written down. The code for the front and backend maintained a proper engineering manner such as low coupled and low coherence. The system is also efficient to be extended with public API in the further.

4.4 Testability

The LPS visualiser has implemented a range of data Parser testing file, which can also be further used by the developer to test. All the testing file has covered the data type of different road, vehicle, program modifier testing, logic testing etc. If the testing file is not enough to test the program, the developer can use the 'lps-generate-spec' CLI function to test the LPS program on a command-line level. The result of testing is shown in Fig. 40



((a)) ((b)) ((c))

Figure 40: The different data parser testing result
Unfortunately the grey is not readable. Also, should explain at least one of the outputs.

5 Conclusion

To summarise, the project ended up with the fully functional implementation of the web based LPS visualiser in JavaScript. LPS visualiser are able to visualise different traffic scenario including one lane road, T Junction, overtaking, cross junction and roundabout. Through the testing of LPS programs described in Section, LPS visualiser is proved to meet the requirement specified in the introduction part. By quickly learning through the user guide and the introduction page on web LPS visualiser was able to be used by a non-programmer.
Along with the implementation, the project has also revisited some of the previous ideas and brought about new ideas. In this chapter, some of the challenges faced during the course of this project are documented, along with possible future work that can be extended from this project.

5.1 Challenges faced

1. Hard to dive deep into interpreter. As describem in the background section, the interpreter is not documented with lenthy function and usless clousures. This make other developer to understand and re-implement the interpreter. Another problem is there is no diect source to source LPS to JS conversion, this make programmer hard to extend and interpreter and debug.

2. Lacking of debugging tool and function.
   The only debuging method is to use the testPrint() command to testing wether the LPS action has fired or not. This make debuging very hard continue when the LPS program is large with different event and actions.

3. Latency of interpreter LPS.js
   Since traffic often envolved various of reactive rules. Checking again them take time. When more and more agent are introduced into the program, the JavaScript interpreter will have more goal tree to descrutive reduced. Hence the program will slow down by that.

5.2 Further work

1. APPLY JavaScript direct API
   As the LPS interpreter is designed. Rather than a directly source-to-source, a complex data parser was used. It make programming more complex hence more data parser are needed to visualise the LPS program rather then direct use the JavaScript data. It would be very helpful for further development if a LPS to JS interface is implemented.

2. Shift the front-end to react.js
   Although LPS.js can run on mobile broswer and web broswer on Linux,windos, unix system. It is still not cross-platform. By applying front-end to reat.js, this project can be easily to shift to mobile app using ReactNative framework which is another light weight mobile app framework. It use react like front end and JavaScript to build mobile apps.

3. Modify the LPS program and push direct onto MongoDB cloud.
   The function of posting self-defined LPS program directly to the MongoDB cloud could be a valuable furtuer work. If this can be done LPS can have a community of LPS leaner and the user are able to post and view other people's work.

4. Multi-threading LPS
   Multi threading envolved the server and client communication. Hence the reactive rule and traffic inforamtion need to be shared in a blackboard. This structure will be more interesting and more fun if it is implemented on the web. However, implement server and client communication on web is very hard because web communication simply run by HTTP/IP protocol. The server-client socket communication are easy to achieve but it is in application level. It is not possible to achieve the web based P2p communication within the time frame given for this project.

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] Jan Wielemaker, Fabrizio Riguzzi, Robert Kowalski, Fariba Sadri, and Miguel Calejo. Using swish to realise interactive web based tutorials for logic based languages, 04 2018.

[4] 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.

[5] Gary Jones and Frank E Ritter. *Production Systems and Rule-Based Inference*, page 177–199. 01 2006.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] Wikipedia,NodeJs. Node.js. `https://en.wikipedia.org/wiki/Node.js`, 2019. Online; accessed 13 Auguest 2019.

[20] ExpressJS. ExpressJS. `http://expressjs.com/`, 2019. Online; accessed 13 Auguest 2019.

[21] searchapparchitecture. RESTful API. `https://searchapparchitecture.techtarget.com/definition/RESTful-API`, 2019. Online; accessed 13 Auguest 2019.

[22] Wikipedia,MongoDB. MongoDB. `https://en.wikipedia.org/wiki/MongoDB`, 2019. Online; accessed 13 Auguest 2019.

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

Appendices

Some Appendix

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

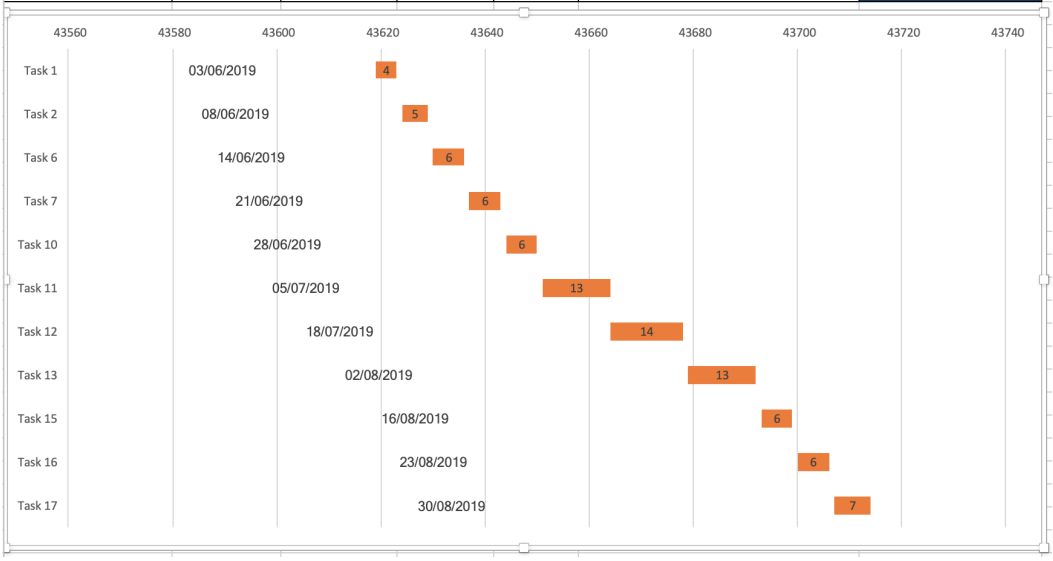


Figure 41: Gantt chart plan: shows the plan that was made to achieve the goal. It was divided into 5 phrases. First of all, a background report need to be written to formalise all the background and the work that I have already done. The first phrase is the background report writing. Second phrase is 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.

Fig. 41