Overview:

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

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

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

1. What front-end tool is going to be used to achieve live animation.

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

1. What information of LPS program need to be prased 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

const LPS = require('lps');

LPS.loadString('...')

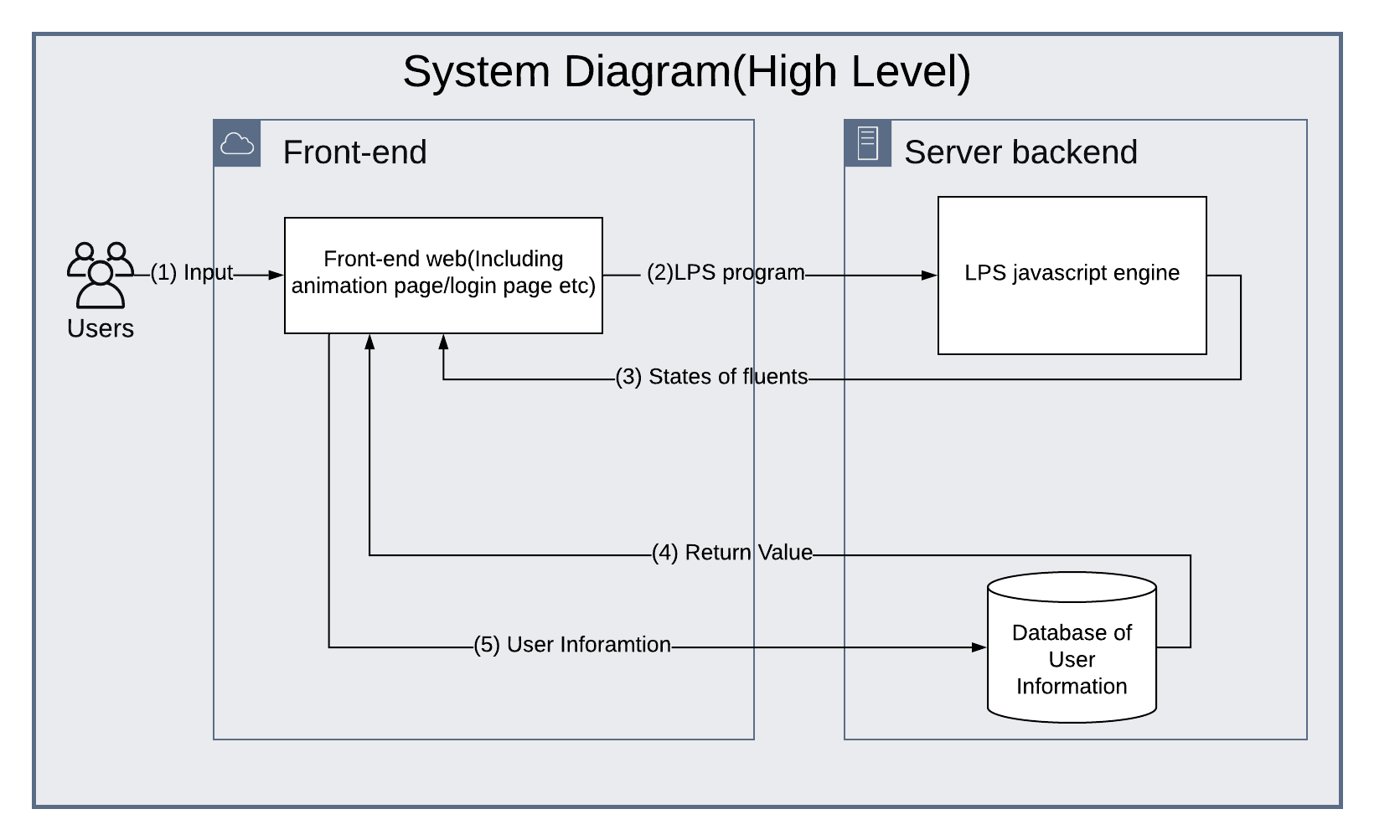
.then((engine) => {

engine.run();

});

The data that we are parse onto the web is then solved. The only issue we need to solve is to formatting different fluents into different data structure and capture the change of fluent in each running cycle, detail will be illustrated in the following subsection.

Structure Design:

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

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 webpage 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.

LPS program

Overview：

The LPS program is the input program. For the traffic the LPS program should not only follow the structure of the interpreter understandable way as we discussed in section (?) also it should be easy to be parsed into the PIXI animation engine. Hence the format is defined as below (User can replace any Upper-case text to real instance number or predicate:

% we assume the destination is reachable

maxTime(Number).

cycleInterval(Number).

fluents([

Set of fluent predicate defined

]).

events ([

Set of event predicate

]).

Observe([

Set of observation predicate (happed 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)

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.

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.

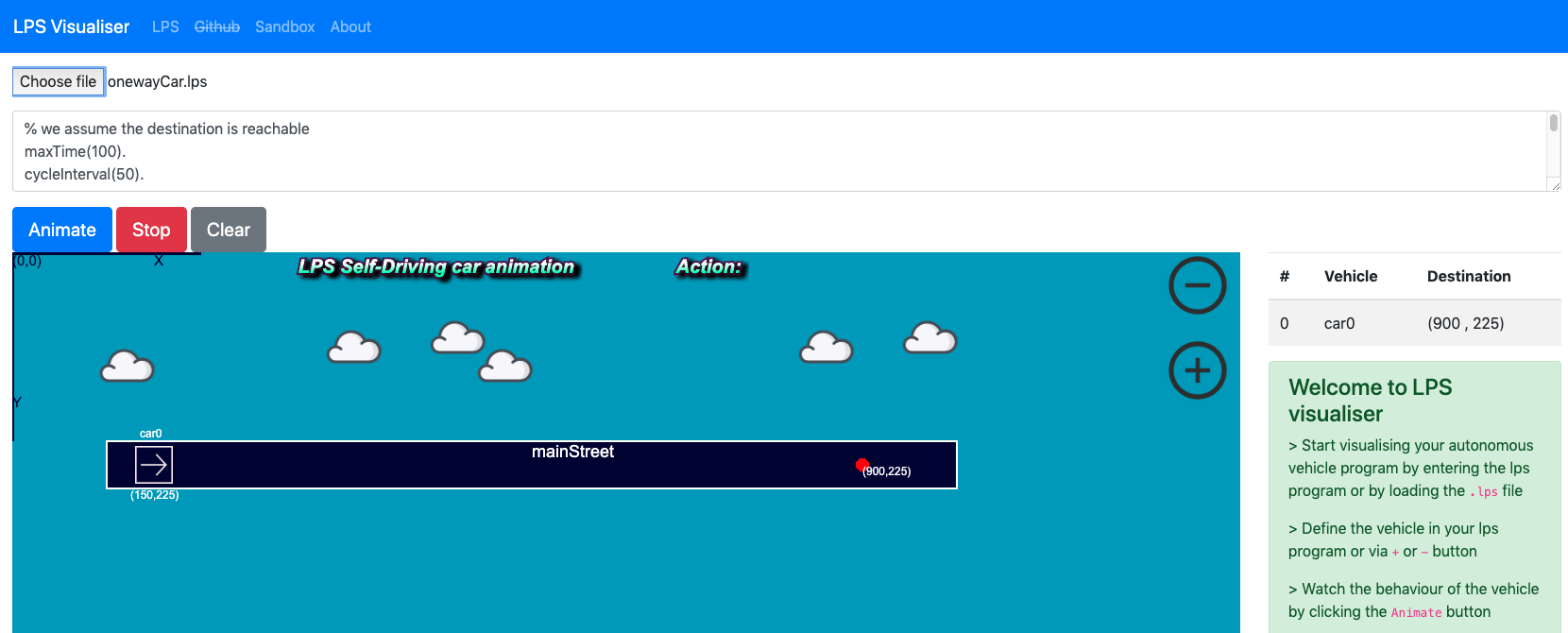
Define traffic light(fluent) as: trafficLight(coordinate(X, Y), Working\_status, Color, FacingDirection) ie: trafficLight(coordinate(430, 475), red, westward, mainStreet) meaning there is a traffic light located at coordinate(430, 475) and it is showing red color at the moment, the traffic is facing

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)

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

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).

Car moving straight



the lps program example is

% we assume the destination is reachable

maxTime(100).

cycleInterval(50).

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

fluents([

stopped(VehicleName),

moving(VehicleName),

coordinate(X, Y),

location(VehicleName, coordinate(X, Y), Direction),

trafficLight(coordinate(X, Y), Working\_status, Color, FacingDirection),

street(StreetName, coordinate(X, Y), Width, Height, Number\_of\_lane),

goal(VehicleName,coordinate(X, Y))

]).

actions ([

step(Vehicle, NextPlace),

turn(Vehicle, NewHeading),

arrive(Vehicle)

]).

initially([

moving(car0),

location(car0, coordinate(150, 225), eastward),

goal(car0,coordinate(900, 225))

]).

street(mainStreet , coordinate(100, 200), 900, 50, 1).

cloud(coordinate(140,100)).

cloud(coordinate(250,90)).

cloud(coordinate(300,120)).

cloud(coordinate(640,100)).

cloud(coordinate(750,90)).

cloud(coordinate(800,120)).

goal(Vehicle,coordinate(A, B)) from \_ to T,

location(Vehicle, coordinate(X, Y), Direction), A==X, B==Y,moving(Vehicle) at T ->

% testPrint(Vehicle+ ' we have arrived'),

arrive(Vehicle) at T.

goal(Vehicle,coordinate(A, B)) from \_ to T,

location(Vehicle, coordinate(X, Y), Direction), A!=X at T ->

% need to find the right direction here

% driving forward

drive(Vehicle) from T to \_.

goal(Vehicle,coordinate(A, B)) from \_ to T,

location(Vehicle, coordinate(X, Y), Direction), B!=Y at T ->

% need to find the right direction here

% driving forward

drive(Vehicle) from T to \_.

drive(Vehicle) from T to T1 <-

location(Vehicle, coordinate(X, Y), Direction),

Direction == northward,

NewY = Y - 10,

NextPlace = coordinate(X, NewY),

step(Vehicle, NextPlace) from T1 to T2.

drive(Vehicle) from T to T1 <-

location(Vehicle, coordinate(X, Y), Direction),

Direction == southward,

NewY = Y + 10,

NextPlace = coordinate(X, NewY),

step(Vehicle, NextPlace) from T1 to T2.

drive(Vehicle) from T to T1 <-

location(Vehicle, coordinate(X, Y), Direction),

Direction == westward,

NewX = X-10,

NextPlace = coordinate(NewX, Y),

step(Vehicle, NextPlace) from T1 to T2.

drive(Vehicle) from T to T1 <-

location(Vehicle, coordinate(X, Y), Direction),

Direction == eastward,

NewX = X + 10,

NextPlace = coordinate(NewX, Y),

step(Vehicle, NextPlace) from T1 to T2.

% on(coordinate(X,Y),Street) <-

updates(step(Vehicle, NextPlace), location(Vehicle, OldPlace, Direction), location(Vehicle, NextPlace, Direction)).

updates(turn(Vehicle, NewHeading) , location(Vehicle, Place, OldHeading), location(Vehicle, Place, NewHeading)).

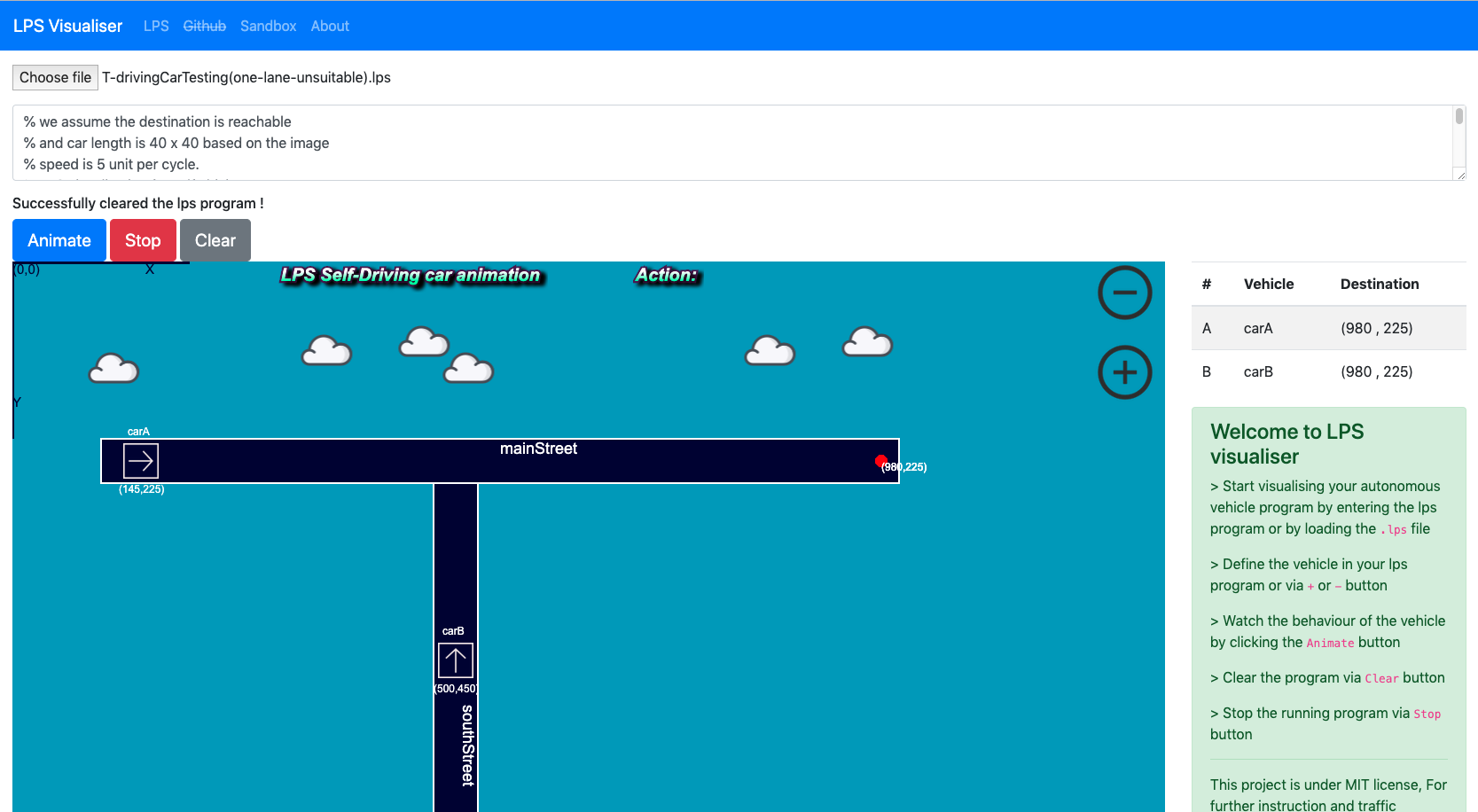
terminates(arrive(Vehicle), moving(Vehicle)).

initiates(arrive(Vehicle), stopped(Vehicle)).

% you can not move (x++) if you are not facing eastward

the basic logic here

T junction with both narrow lane.



T junction with one narrow lane and a double lane

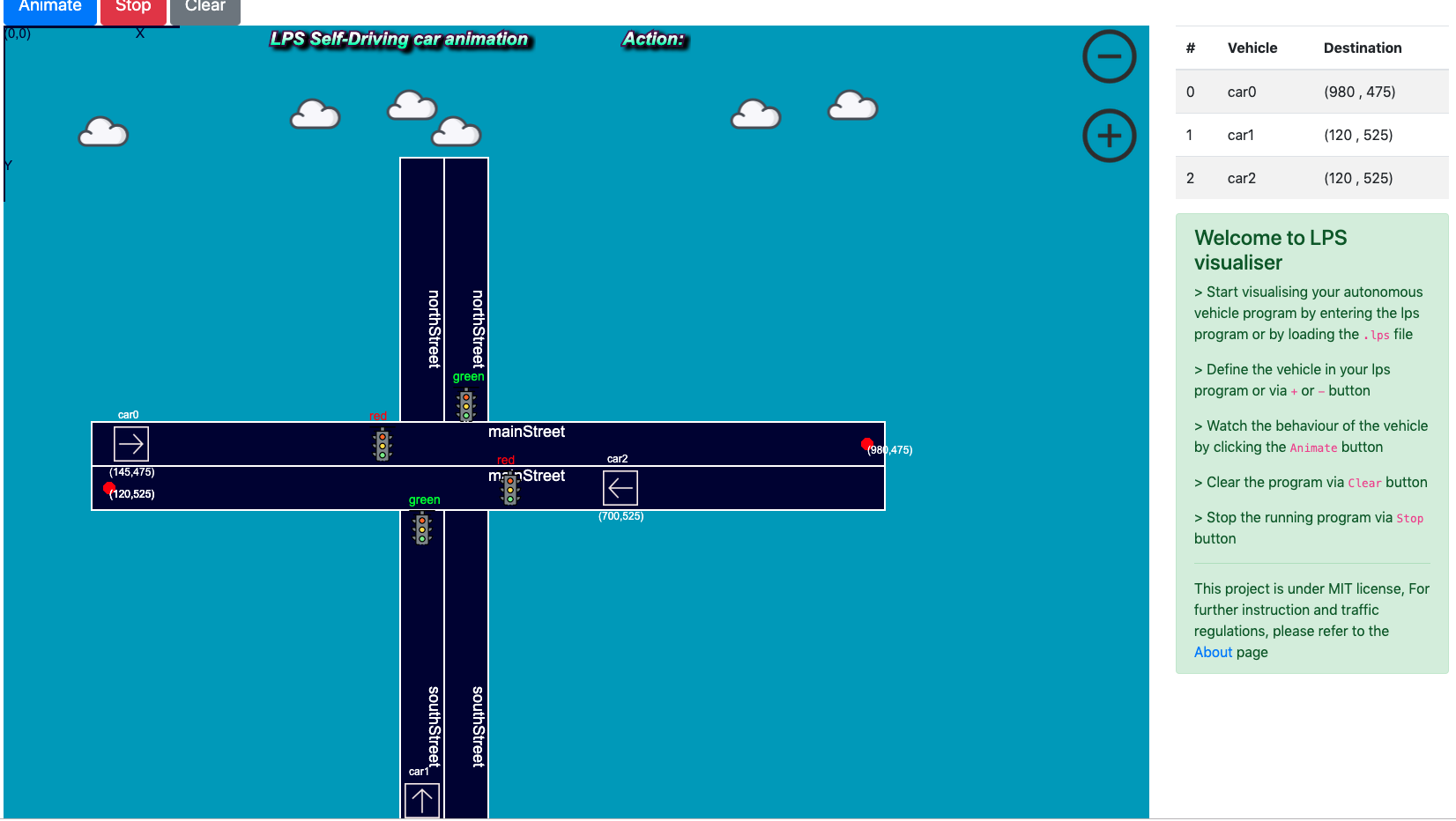
Two car:



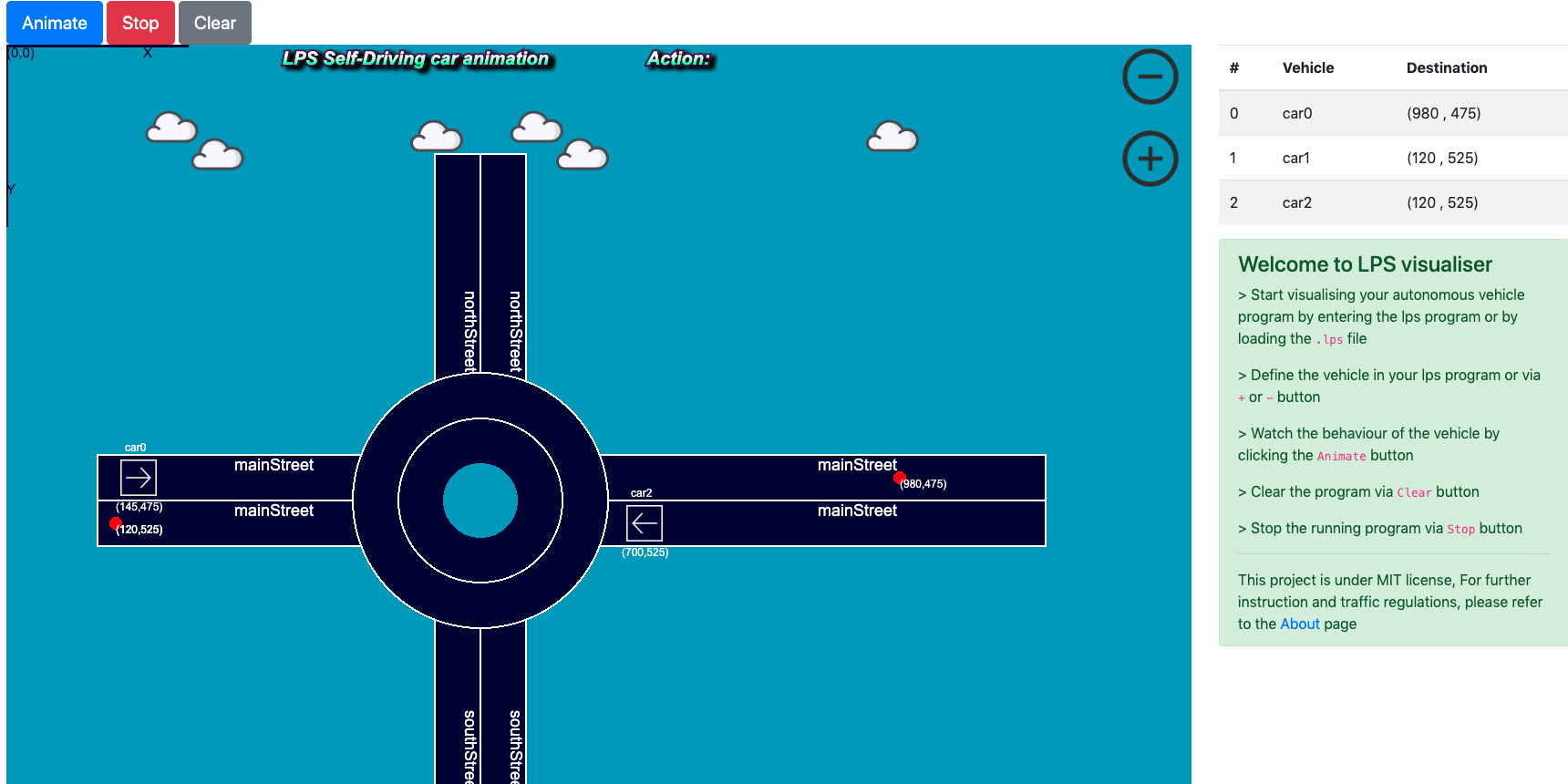
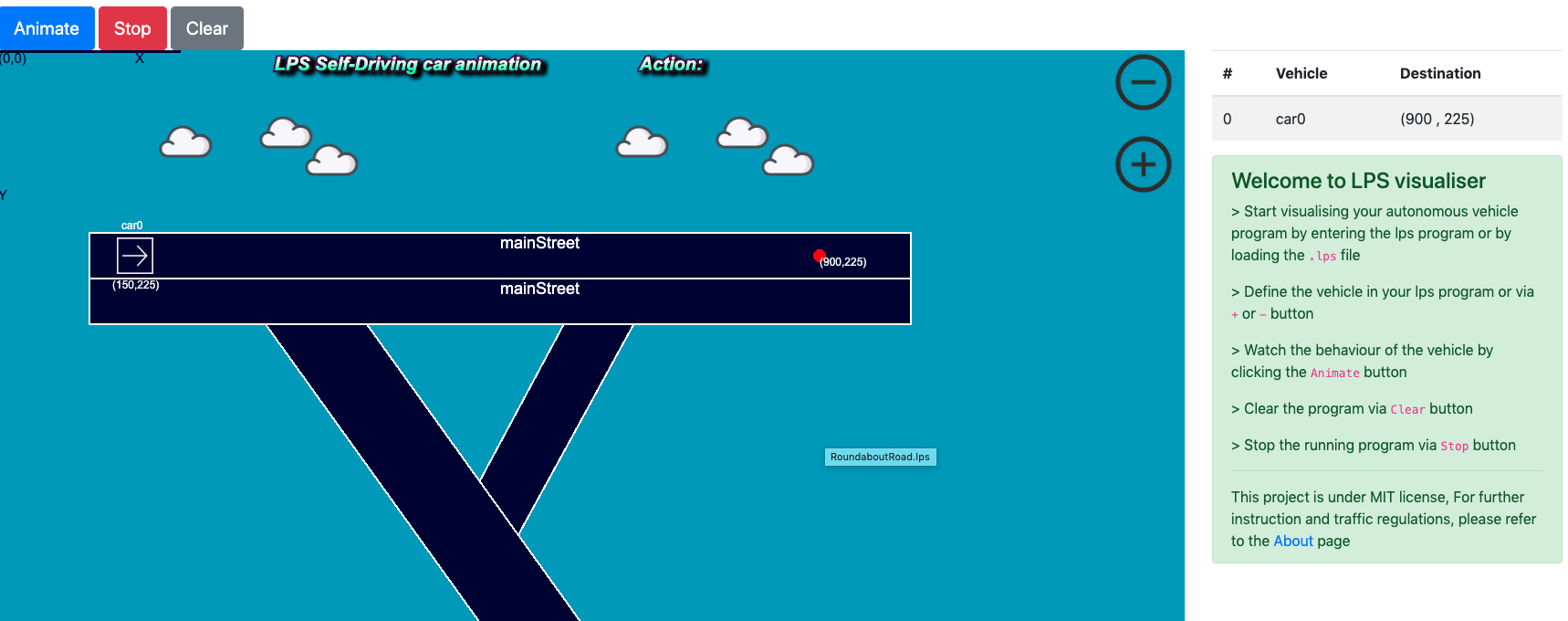
Three cars

T junction with two double lanes

Cross junction with traffic light

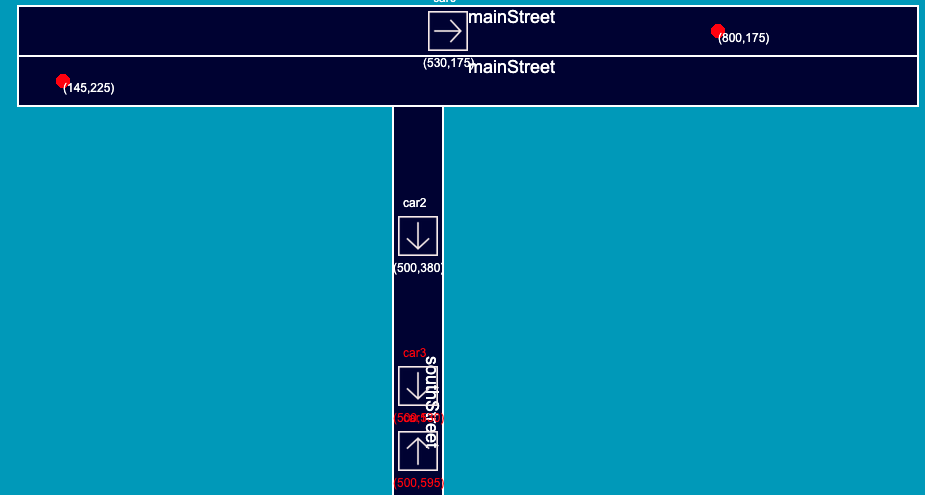


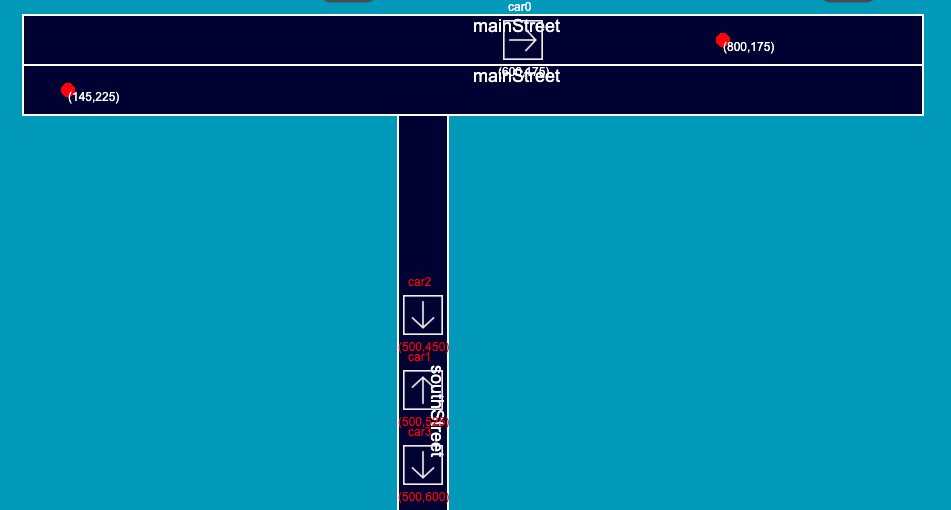
Other Traffic road interface:

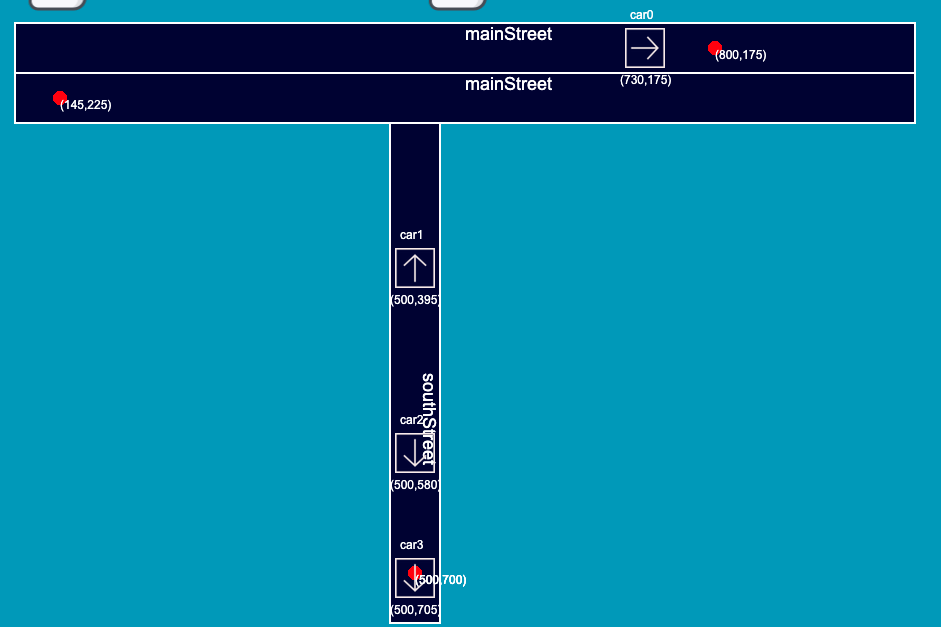


Traffic with restart feature

Cars passing through each other on a narrow road







Overtaking