User's Guide: Traffic Simulation Program

Overview

The Traffic Simulation Program is designed to display time, traffic signals, car position and speed, and other information for traffic analysts to utilize in their work. The system has been designed to be extremely flexible and allows an unlimited number of signals and cars to be created within the parameters of the CMSC 335 Final.pdf document.

Setup

Prerequisites:

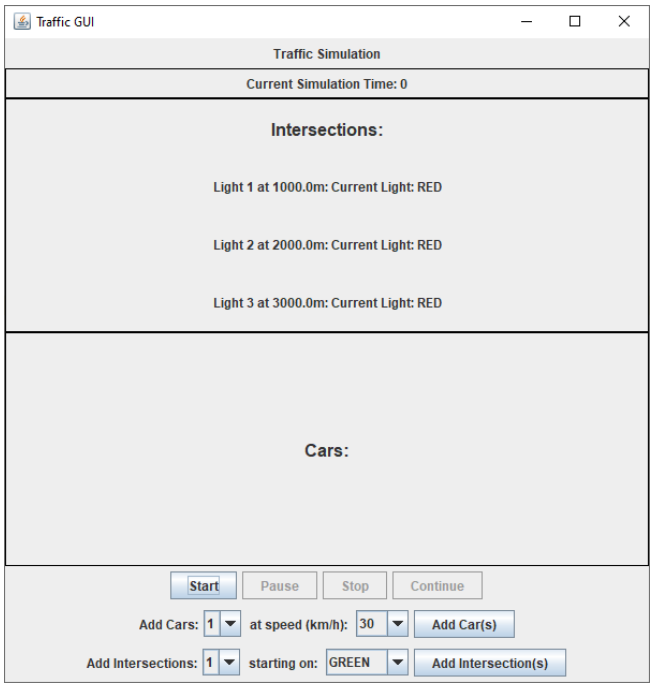
Ensure you have a Java Development Kit (JDK) installed, preferably JDK 8 or above.

Steps:

1. Download the Traffic Simulation Application source files.
2. Store them in a directory of your choice.
3. Using your preferred Java IDE (e.g., Eclipse, IntelliJ IDEA), import the files into a new Java project.

Running the Application

Launch TrafficSimulator.java and an GUI will open up (as seen on the right).



There are three sets of controls:

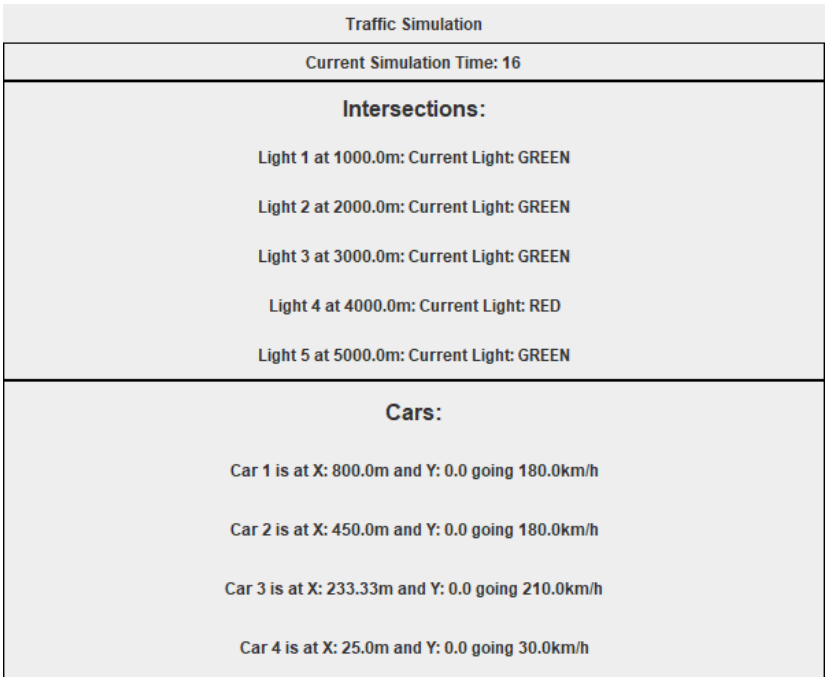
1. **Simulation Controls**: These buttons allow you to Start, Pause, Stop, and Continue (unpause) the simulation. The buttons will become grayed out if they are unusable at the current simulation state.

2. **Add Car Controls**: This panel allows you to add between 1 and 3 cars each going between 30 to 210 km/h to the simulation. All cars will start at (0,0) and will move while the simulation is running, stopping at red lights.

3. **Add Intersection Controls**: This panel allows you to add between 1 and 3 intersections each 1000m farther than the last. You may also choose a starting active

light (Green, Yellow, or Red) for the created lights. All lights will switch from Green to Yellow to Red while the simulation is running.

While the simulation is running, you may utilize the Intersections and Cars panels to view real-time information on the objects of interest.



Design Philosophy and Class Explanation:

Philosophy: This program utilizes a “Model-View-Controller” (MVC) design where the Controller object acts as an intermediary between the GUI (View) and calculation class (Model), taking in user actions on the GUI, sending them to the model to be calculated and then taking the changes from the model and updating the GUI.

TrafficSimulator: Creates the initial JFrame and creates instances for our three main classes:

GUIDisplay: Builds all panels within the initial JFrame and accomplishes all GUI-related logic. Displays relavent information such as simulation timestamp, intersection status,

and car information. Contains all program controls in the form of buttons and dropdowns.

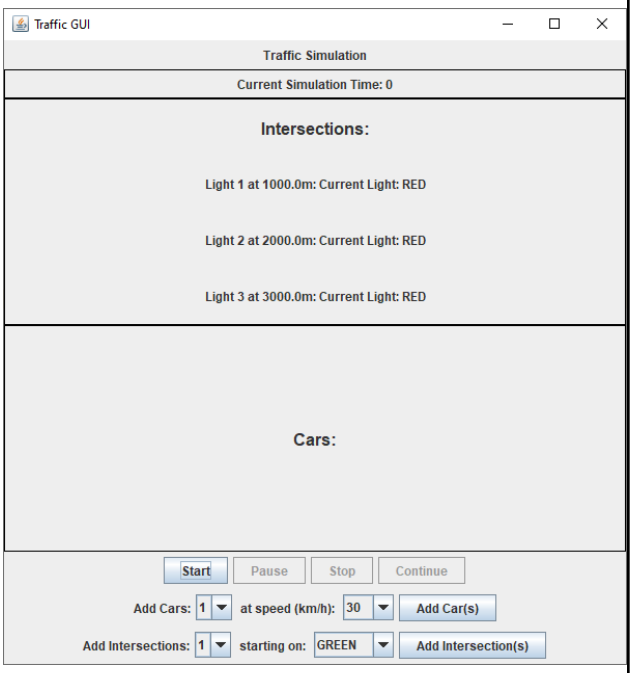
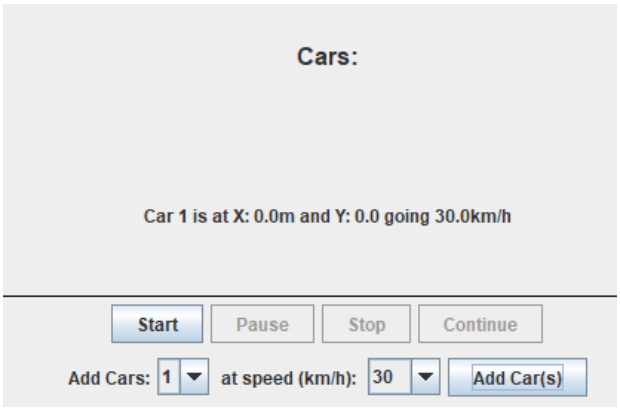
TrafficCalculator: Creates master ArrayLists for all TrafficLight and Car objects that are existent within the simulation. Does all calculations for changing traffic lights and car movement utilizing the internal methods of those objects.

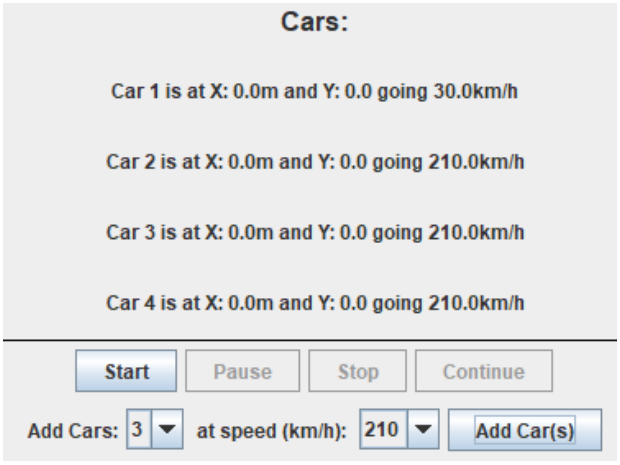
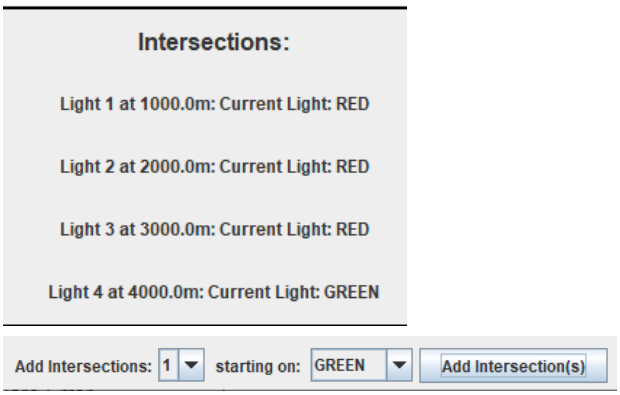
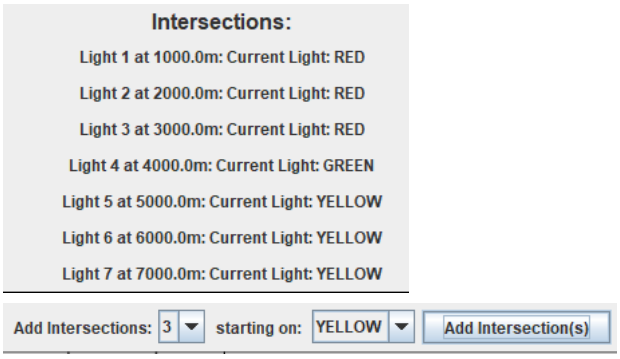
SimulationController: Controller object that controls the thread logic and interfaces between GUIDisplay and TrafficCalculator. Sets ActionListeners on button objects of GUIDisplay that execute anonymous and lambda functions to control the simulation playback and to add cars and new intersections to the simulation. When simulation is paused by “Pause” button, the simulation is halted without additional updates until the “Continue” button is pressed and the simulation updates normally. When simulation is stopped by “Stop” button, the simulation is reset back to the initial starting conditions upon pressing “Start”.

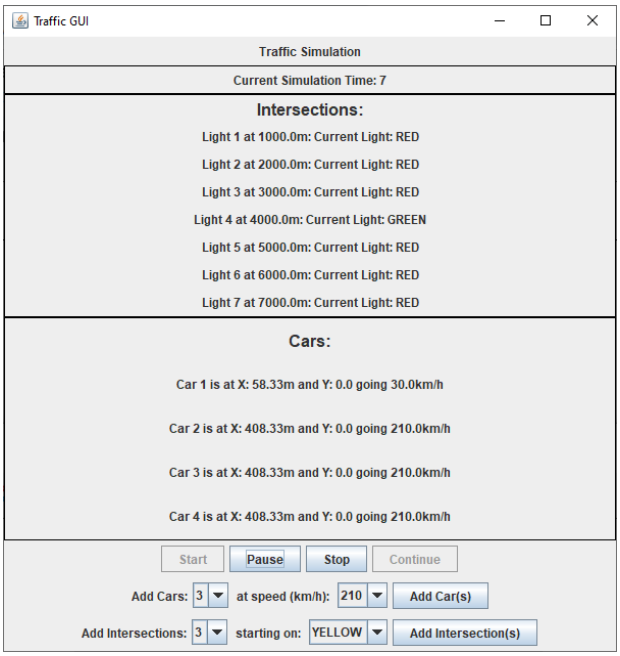
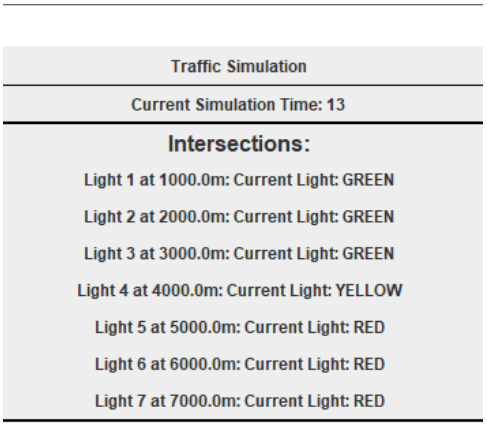
TrafficLight: Object class that holds all attributes and methods needed to simulate a traffic light within the simulation

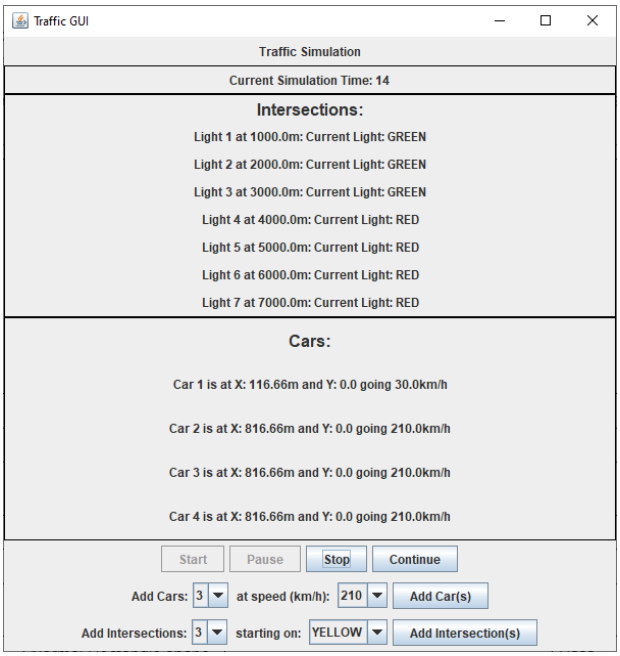
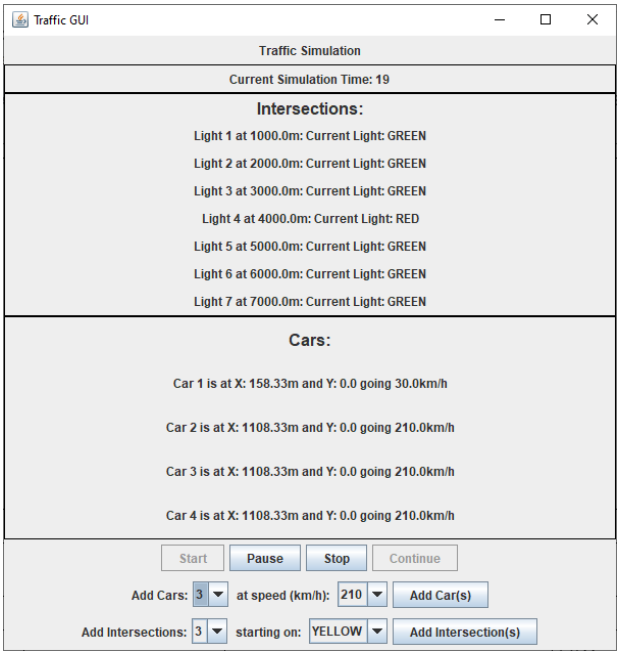
Car: Object class that hold all attributes and methods needed to simulate a car within the simulation.

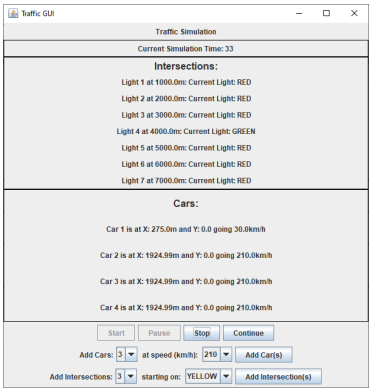
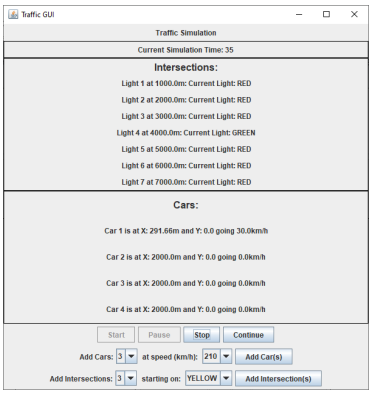
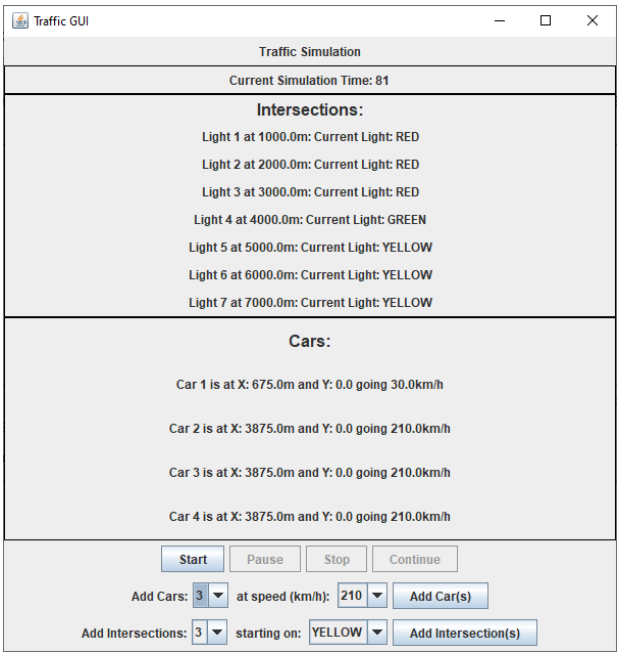
Testing:

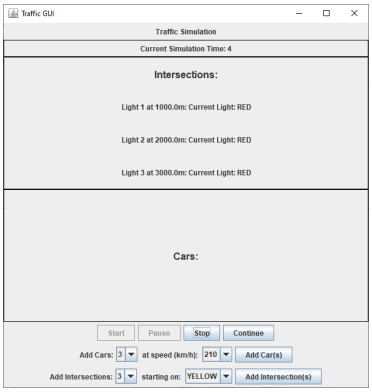
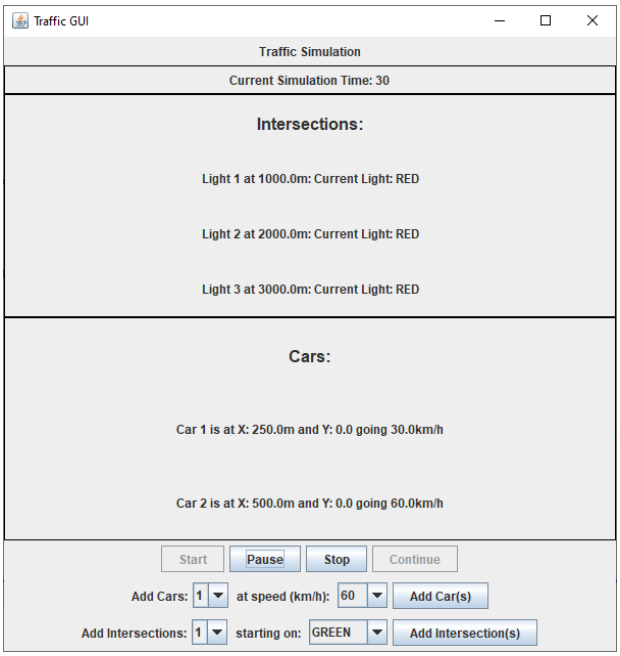
Test #	Description	Screenshot	Pass/Fail
0	Start Program		Pass
1	Add 1 Car (30km/h)		Pass

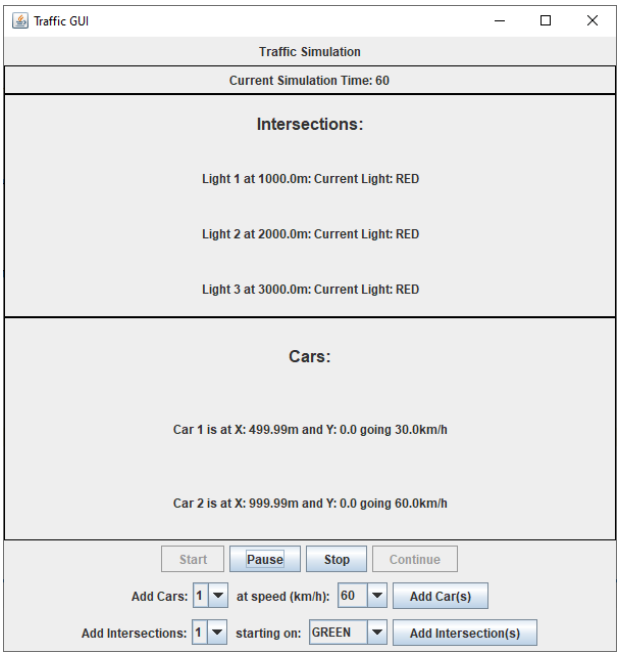
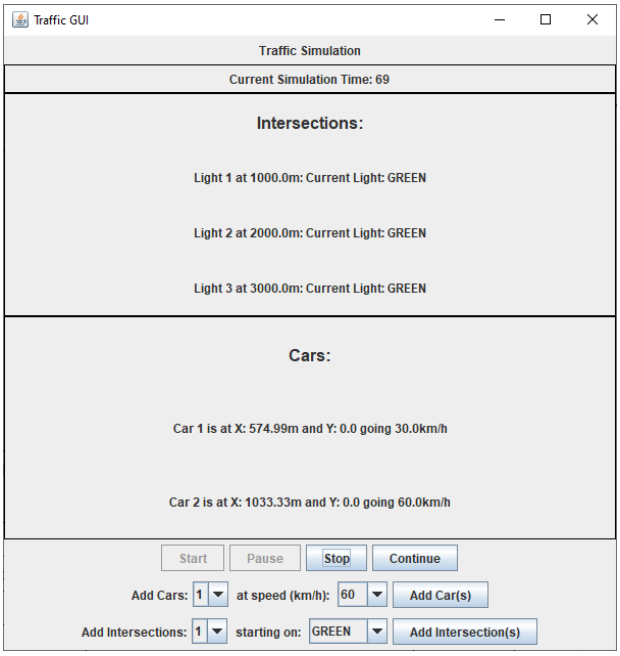
2	Add 3 Cars (210km/h)		Pass
3	Add 1 Intersection (Green)		Pass
4	Add 3 Intersections (Yellow)		Pass

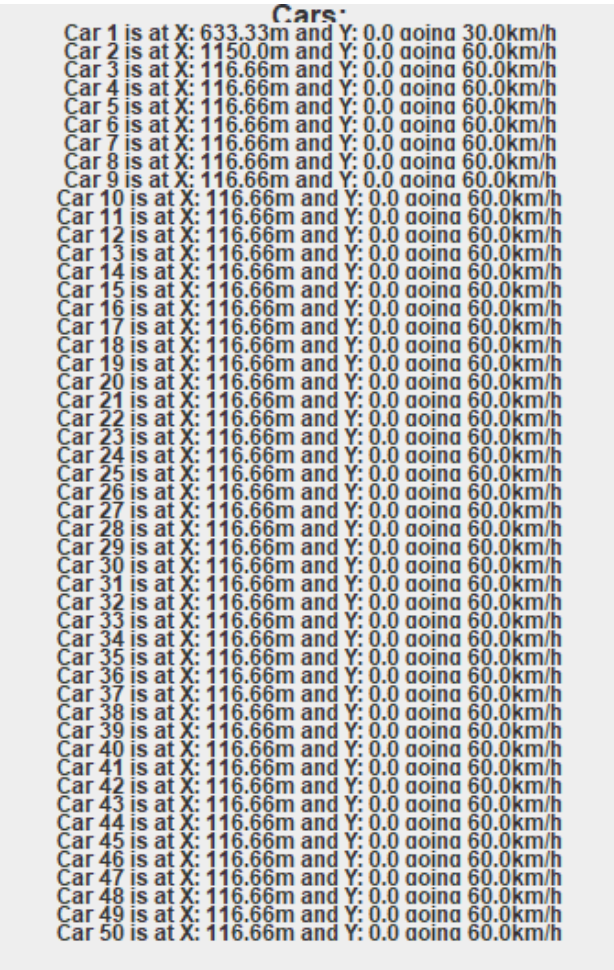
5	Start Simulation		Pass
6	Lights Change by 12 seconds		Pass

7	Simulation Pause		Pass
8	Simulation Continue		Pass

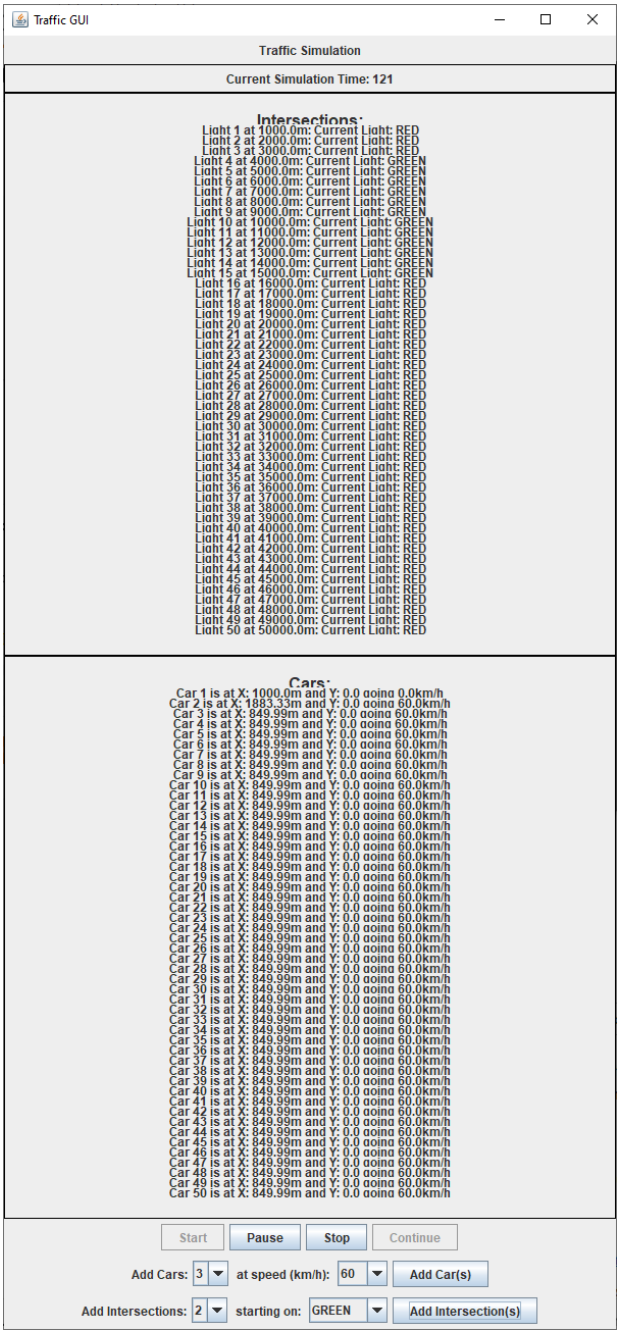
9	Cars stop at red light location (Top: Before reaching red light; Bottom: Stopped at red light)	 	Pass
10	Simulation Stop		Pass

11	Reset on Simulation restart		Pass
12	Cars move at appropriate speed (30s = ½ speed/60)		Pass

13	Cars move at appropriate speed (60s = speed/60)		Pass
14	Cars do not stop at yellow or green light location (Car 2 has passed Light 1 once it turned green)		Pass

15	50 Cars added		Pass (simulation still runs)

16	50 Lights Added		Pass (simulation still runs)

17	Screen resize fit		Pass
18	Normal Exit	*** Welcome to the Traffic Program! *** *** Thank you for using the Traffic Program! *** *** Logging out at 21:46:35 2023/10/09 *** --- BUILD SUCCESS --- Total time: 05:27 min Finished at: 2023-10-09T21:52:01-06:00 ---	Pass

19	ALT+F4 Exit	*** Welcome to the Traffic Program! *** *** Thank you for using the Traffic Program! *** *** Logging out at 21:52:25 2023/10/09 *** -- BUILD SUCCESS -- Total time: 15.728 s Finished at: 2023-10-09T21:52:40-06:00 -- '	Pass

Lessons Learned:

This was a challenge worthy of being the final project in a 300-level college course. I actually had to create this program twice: My initial attempt contained only a single major class that created the GUI, took in user inputs, and attempted to control the threads of the program. I was able to get this class to build a baseline GUI and add TrafficLight and Car objects to the ArrayLists, but thread control and a updating the GUI display for the new objects just never really worked right due to the unwieldy nature of such a mammoth class. Eventually, I decided I was going about it the wrong way, so I did some additional research on how others have tackled this problem. This lead me to the Model-View-Controller design and I immediately understood that was a better way to accomplish what I wanted to do.

When I rebuilt the program in MVC style, I was able to reuse some of my code, but the logic that controlled the interface between the new major classes had to be completely rewritten. I learned the power of encapsulation while doing this, though, because for each major class I mocked up a basic text response system to ensure each of my systems were working correctly (for instance, a println() that showed the car's position every 60 cycles at a set speed). This prototyping was an incredible tool that enabled me to actually compile and run my completed program with *no errors* the very first time I attempted. I will absolutely be planning a prototype phase like this again in the future.

Finally, I want to touch on the thread control aspect of this program. My initial attempt to assign logical items to individual threads within my single mega-class was a disaster where I was attempting to watch objects take actions and then assign the threads, but objects were getting updated at seemingly random and there was basically no interleaving or concurrency happening. Once I split the classes in the MVC and had the Controller class handle everything with threads, it was much simpler to just assign two system flags (isPaused and isRunning) and create a single intrinsic lock object (pauseLock) that were completely arbitrary to the execution. I referred back to the reading of the past two weeks many times to make sure I was utilizing these basic concepts correctly, but it made a world of difference by separating out the thread logic that way!

Overall, this is probably the most complex program I've ever written, but seeing the interplay of cars stopping at lights, lights switching on their own, and the GUI actively responding to my inputs is a real joy, so it was definitely all worth it. My main take aways are: 1. Focus on encapsulation even more than I think I need to; 2. Separate out thread logic and make it controlled by arbitrary objects, if possible; 3. Prototype out each portion of code before trying to link it all together.