Kevin Peng
Greg Bischoff
Jessica Yue

CMSC240 Intersection Final write-up

Class design

We followed an object-oriented approach. We created objects of the necessary parts and put everything together in a “main” method, in our case, involved the SimTest.cpp file. The classes involved are listed:

Classes used: (Indentations signify dependency relationships, SimTest Exception)*

	Vehicle		- Cars, trucks, SUVs, lengths, movement intent
		Random	- Used to generate numbers to create random Vehicle types and movement intent.
			- Worth noting that the parameters that this use, the seed and such, are static. They are set in the SimTest.cpp file because that is where these methods are used to construct the objects..
		Probability Parser	- given input file, construct proper probabilities of vehicle type and movement intent.
	Stoplight		- Red, yellow, green, countdown
		Light Parser	- given proper input file, sets the times of lights.
	Lane		- Creates a lane of Sections where vehicles will be able to reside.
		Section	- Used for each “unit” of lane. Pointers to the vehicle
**	SimTest		- This is what we used as our “main.” Creates an instance of Sim and has an update method
		Sim	- Simulation class. Constructs all of the vehicles, lanes, stoplights, and the intersection and runs the simulation.
			- This file requires all of the object files as dependencies.

Testing Protocol:

Our approach to testing involved creating tester files. For example, to test Vehicle, we created a VehicleTester.cpp class. This class includes the necessary dependencies, which are Random and ProbabilityParser classes. It would then generate a vehicle and test our getters, printing out the values involved. Each object file ultimately had some tester file, which would test for edge cases and, if necessary, tested proper input files extensively.

Kevin Peng
Greg Bischoff
Jessica Yue

<u>How to use:</u>

In order to use our file, you should use “make” which will create *.o files and compile the program. In order to modify input parameters values, you will have to open the “input.txt” and “lights.txt” files and change the numbers. Both of the *.txt files follow the format of “(type)”=”(number)” where the “types” are the variables that will be initialized to “number.” In terms of the input.txt, the numbers can be any integer values and the way it works is that the number of cars, SUVs, and the truck = totalVehicles, and to find the probability, the random class creates a random.number between 0 and the totalVehicle number. If the random.number falls between 0 - (cars.number), it will be a car. If the random.number falls between (cars.number) - (SUVs.number), it will be an SUV. If it’s neither of those, it will generate a truck. The same process follow for generating probabilities of the movement types. The lights.txt just contains the times of green and yellow lights, where red = the sum of green and yellow times, so it should always work.

After generating those, on the command line, run “./main” and if you include another argument, like “./main 400,” it will run through the simulation however many ticks you put, 400 ticks in this example. Otherwise, if there’s no argument, it will run 100 times by default.

In order to run the project step by step, it is necessary to press “Enter” after each run, signifying that it should update and move everything one tick. The probabilities of total vehicles moving through the intersection will print out to console at the end.

A note about the probabilities, the way the cars are made, it must have a single blank space in order to conditionally make. For SUVs, it will conditionally make it only if the conditional make for car has failed once, which is equivalent to two blank spaces. For Trucks, it will conditionally make, only if the conditional make fails for car and SUV once ,equivalent to 3 free spaces. Therefore, the statistics at the end, if the given inputs are all the same, the expected statistics will be that the trucks will be half of SUVs, and SUVs will be half of cars.