ELEC-A7151 Object oriented programming with C++
Software project - Traffic Simulator
Project Documentation

Hatanpää Väinö, Herold Adam, Kelekar Aditya, Selvarasa Gowshigan

December 11, 2020

1 Introduction

For our software project we have created a traffic simulator in C++. It is an easy-to-use tool for generating a simple but realistic simulation of a city with car traffic and passengers driving to work, home or to the shop.

This tool cannot be used for simulating a specific real city or situation, since the city layout is pseudo randomly generated, but it can be used to get a more general look into the city traffic problematic.

2 Overview and Features

2.1 City Generation

The program features an automatic city generation with variable size of the city and number of passengers (inhabitants). The user can change the input by editing the files *input_file*. The first, second and third lines correspond to the size of the city, number of passengers and the density. The generated city tries to achieve a simple but realistic layout.

The city is generated into a JSON file, which can be saved for later usage.

2.2 Traffic Simulation

During the simulation of one day, we can see a realistic behavior of people going to work in the morning (normal distribution around 8 AM) and going back home (normal distribution around 4 PM) and going to a shop at random times during the day.

All these aforementioned trips are taken by car. Cars move from one building (e.g. home) to another (e.g. workplace), taking the shortest path (calculated using the BFS - Breadth-first search algorithm).

2.3 Analytics Tool

The program offers a plotting function which can be used to produce a bar graph with the number of vehicles that had driven through a road each hour of the day.

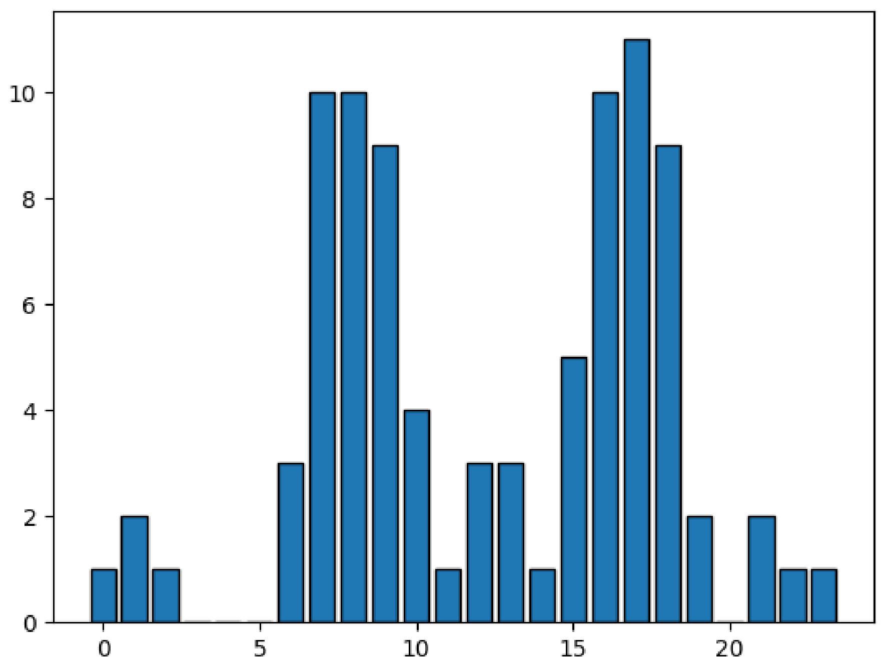

Figure 1: An example bar graph produced by the program.

3 Software Structure

3.1 Main classes

- **Building**: Buildings are the destinations passengers are going to. There are different types (residential, commercial, industrial).

- **Vehicle**: Vehicles are used by passengers to get from on building to another.

- **Main and Side Road**: Roads always connect two points (buildings or intersections).

- **Intersection**: Class representing a point connecting multiple roads. Intersections are also used as nodes in BFS when vehicles search for a path.

- **Passenger**: Passengers act accordingly to their schedule by taking actions at appropriate times (go to work, go home, go shopping). They drive vehicles to go from building to building.

- **City**: A container class used to store all the other objects in one place.

- **Map**: Class used for rendering graphics via SFML. It gets data from the city object.

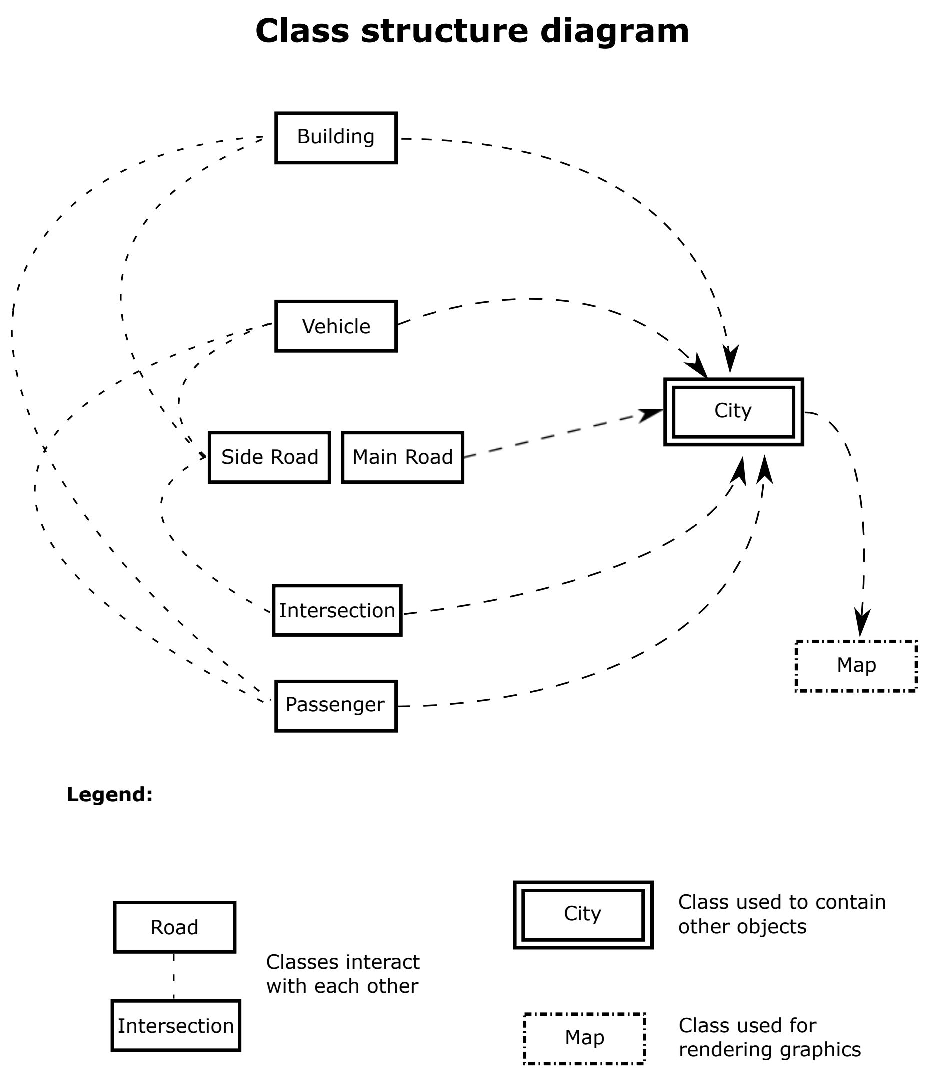

Figure 2: Diagram illustrating the basic structure of the main classes.

4 Use of external libraries

- **SFML**: Simple and Fast Multimedia Library, is a portable API written in C++. It is composed of five modules: system, window, graphics, audio and network.
- **matplotlib-cpp**: Visualizations for analytics
- **nlohmann/json**: Used for parsing the input file and saving the generated city layout.

In addition the third-party libraries, we also need a X11 server. In this project, we have used the *vcxsrv*. It can be downloaded from https://sourceforge.net/projects/vcxsrv/ and installed. In addition to that, *xfreerdp*, a X11 remote desktop client protocol will be usefull to diagnose if there is communication error with the X11 server. For example, one could try *xfreerdp ..* command to see the whether X11 is running and find out the host address. Before running the software user has to set the *DISPLAY* variable in terminal. Execute the following command in order to set the variable.

```
export $DISPLAY=:0
```

If you are using WSL, what worked for us was instead this command:

```
export DISPLAY=:localhost:0.0
```

Alternatively user can set this variable by default by editing the *.bashrc* file in the home directory. After setting, execute the following command to update the variable. if the suggested variable for display does not work, user may use the *xfreerdp ..* command to find the current display variable.

```
source ./bashrc
```

Once set the environment variable we are set to proceed to the next step.

5 Building and Using

The user need to install third-party libraries before start using this project. We have used *cmake* as build processing and managing system. It is expected the user already have cmake, if not the cmake can be installed in following the instruction show in the url. https://cmake.org/cmake/help/latest/command/install.html

1. SFML: The sfml library can be installed in linux following the this link https://www.sfml-dev.org/tutorials/2.5/start-linux.php. For example, in debian, one could execute the following command

   ```
   sudo apt-get install libsfml-dev
   ```

2. matplotlib-cpp: This library can be installed following the steps given below

   ```
   sudo apt-get install python3 python3-dev python3-pip
   pip3 install matplotlib
   ```

   If you are using WSL you might need to install one additional package with the command

   ```
   sudo apt-get install python3-tk
   ```

5.1 Compiling

Firstly navigate to the project directory. Then, create a folder named **build** and follow these steps. The instructions are also given in the *README* file.

```
mkdir build
cd build
cmake ..
make
```

The success-full execution of the command *cmake* verifies that the third-party libraries are correctly installed and we can go to compilation directly. Next we can start the compilation process. We have named the executable as *traffic-simulator*. Execute the following command in the build directory to run the software.

```
./traffic-simulator
```

5.2 User Guide

- **City generation and user input**: When the program is ran without any user input arguments, a city will be generated, based on the variables set in the file `src/input_file.txt`. This file contains two variables the user can set:
  - `Passengers X`, where X is an integer from 1 to 1000, sets the number of inhabitants and their cars.
  - `Size Y`, where Y is an integer from 1 to 100, sets the relative size of the city.

  Beware that setting both input variables to their maximum value will result in a very slow simulation (one day takes about an hour).

  The generated city (the layout, passengers, etc.) is saved to a JSON file called `input.json` which will be automatically stored in the `build` directory (the directory with the executable program). This JSON file can be saved by the user (renamed, copied) and can be used as an input to the program itself.

  Let's say we generated a city layout we would like to run multiple simulations with. We rename the `input.json` file as `city1.json` and when we want to run the simulation using this city we simply run the program with `./traffic-simulator city1.json`

Using faulty input files (both the TXT and JSON files) can result in them not being accepted by the program in which case the program uses some default values. If this happens you will see a message in the terminal.

- **General usage**: After starting the simulation by using one of the commands mentioned above, a window displaying the city will pop up. In the window the user can see several things:
  - The roads in white colour. Bigger roads connect intersections and can produce a histogram of cars (more info below) and the smaller ones connect buildings to intersections.
  - Buildings in orange colour.
  - Cars driving on the roads going from one building to another. Cars have three different colours depending on their destination: green cars are going to a shop, red cars are going to work and blue cars are going back home.

  In the terminal user can see the time pass in the simulation.
- **Analytics**: User can make a histogram of average cars on the road per hour of the day by clicking any main road. This graph can then be exported as a picture from the GUI.

6 Testing

Parts of the program have been tested with some sample values and seeing if the output is as expected. Then the program has been tweaked until the result is right. For example for analytics sample input was an array 1,2,3,4,3,2,1 and the output was a correct bar graph. This same principle could also be used for graphical user interface. It was tested with drawing a rectangle and printing a console message if it was clicked.

The user input was tested with different ways of trying to produce an error, to achieve as user-proof program as possible.

The city generation algorithm, cars path finding and passenger logic all worked as intended with many runs of the program.

7 Work Log

Here we present the division of work between the members of our group. Each member also describes their work done each week and the number of hours spent.

- **Hatanpää Väinö**:

Week	Description of work	Hours spent
44	General project arrangements and project plan base	3
45	looking into matplotlibcpp, planning and getting it to compile	7
46	Sample plotting and data analytics planning	4
47	Working on local issues with SFML and X server	4
48	Integrating completed parts of the project to eachother and modifying main loop	6
49	Implementing graphics and large modifications to Vehicle logic and main loop	15
50	Project documentation and polishing a bit of everything	13

- **Herold Adam** has worked mostly on the classes and on the city generation (input processing) part, also took part in the car driving logic implementation.

Week	Description of work	Hours spent
44	The project plan - creating figures and writing some sections.	5
45	Classes - creating classes.	6
46	Input processing - generating the JSON file containing information about the city.	5
47	Input processing - polishing the algorithm for generating the city layout, creating objects based on the JSON file.	8
48	Input processing - finishing and debugging. Main loop - creating basic skeleton for the main loop.	6
49	Adding more features to the classes, fixing bugs.	4
50	Project documentation. Cars pathing. Passenger logic. Debugging and polishing.	21

- **Kelekar Aditya**: Limited participation due to personal reasons. Participated in the planning stage by reviewing the project plan and editing it.
- **Selvarasa Gowshigan**

Week	**Description of work**	**Hours spent**
44	The project plan - writing scope, planned schedule and some paragraphs in other sections	3
45	Classes - map, vehicle, node, path.	10
46	cmake file and sfml libraries adding to the project	3
47	project documentation writing	2