Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

A Mini Project Report
on
"Traffic Simulation System"

[Subject code: COMP 202]
(For partial fulfillment of Year II / Semester I in Computer Science)

Submitted by
Miraj Sapkota (46)

Submitted to
Rupak Ghimire
Department of Computer Science and Engineering

February 28, 2025

Abstract

This report presents the development of a traffic simulation system implemented using C++ and the SFML library using queue data structure. The system simulates vehicle movement across a road network, controlled by adaptive traffic lights. It examines each code file, detailing its functionality, key components, and the approaches I used to address challenges encountered during development.
Keywords: *Traffic Simulation, SFML, C++, Vehicle Movement, Traffic Lights, Lane Management*

Bona fide Certificate

This project work on

“Traffic Simulation System”

is the bona fide work of

“

Miraj Sapkota (46)

”

who carried out the project work under my supervision.

Subject Instructor

Rupak Ghimire
Visiting Faculty
Department of Computer Science and Engineering

Contents

List of Figures

Chapter 1: Introduction

This project focuses on developing a traffic simulation system using C++ and the SFML library. The system models vehicles moving through lanes and intersections, regulated by traffic lights. In this report, I outline the design of each component, explain the purpose of each file, and discuss the challenges I faced and resolved during development.

1.1 Objectives

1. Develop a traffic simulation system using C++ and SFML.
2. Simulate vehicle movement through lanes and intersection using queues.
3. Implement adaptive traffic lights to regulate traffic flow.
4. Visualize the simulation using SFML.

1.2 Tools and Methods

I selected specific tools for this project based on their suitability for the task. Below, I explain my choices.

1.2.1 C++

I chose C++ because an object oriented solution seemed natural to me instead of using C and functional approach.

1.2.2 SFML

The Simple and Fast Multimedia Library (SFML) was selected for handling graphics. It is a straightforward method to display vehicles and traffic lights. Compared to alternatives like SDL, SFML is similar but popular among C++ developers like SDL is for C.

1.2.3 CMake

I opted for CMake to streamline the build process Manually writing makefiles was complex, but CMake simplifies this by allowing me to define files and dependencies, such as SFML, in a single configuration.

1.2.4 TCP Sockets Over File Sharing or IPC

I decided to use TCP sockets to facilitate communication between `traffic_generator` and `simulator`. Sockets enable direct and efficient data transfer over a network, ensuring minimal delay in sending vehicle information.

Chapter 2: Design and Implementation

This chapter describes the design of my traffic simulation system. I begin with the enums used throughout the project, followed by a detailed explanation of each file, including its purpose, Attributes, methods, and my implementation approach. Additionally, I provide algorithms for `traffic_generator` and `simulator`, along with examples of a vehicle's lifecycle.

2.1 Enums

Enums define fixed values in the project, ensuring clarity and consistency in the code. Below are the key enums utilized.

2.1.1 Lane

The `Lane` enum specifies the lanes in the simulation:

- `A1`, `A2`, `A3`: lanes on Road A.
- `B1`, `B2`, `B3`: lanes on Road B.
- `C1`, `C2`, `C3`: lanes on Road C.
- `D1`, `D2`, `D3`: lanes on Road D.

These values identify the starting and ending points of vehicles. Also lanes 2 and 3 are outgoing lanes and lane 1 is an incoming lane for vehicles on each road.

2.1.2 RoadType

The `RoadType` enum defines the four roads:

- `A`: Northbound road.
- `B`: Southbound road.
- `C`: Eastbound road.
- `D`: Westbound road.

This enum associates lanes with traffic lights for control purposes.

2.1.3 Route

The `Route` enum specifies possible vehicle paths:

- `MOVE_A_TO_B`: Road A to Road B.
- `MOVE_A_TO_C`: Road A to Road C.
- `MOVE_A_TO_D`: Road A to Road D.
- `MOVE_B_TO_A`: Road B to Road A.
- `MOVE_B_TO_C`: Road B to Road C.
- `MOVE_B_TO_D`: Road B to Road D.
- `MOVE_C_TO_A`: Road C to Road A.
- `MOVE_C_TO_B`: Road C to Road B.
- `MOVE_C_TO_D`: Road C to Road D.
- `MOVE_D_TO_A`: Road D to Road A.
- `MOVE_D_TO_B`: Road D to Road B.
- `MOVE_D_TO_C`: Road D to Road C.

These paths determine the waypoints vehicles follow.

2.1.4 Light

The `Light` enum defines traffic light states:

- `RED`: Stops vehicles.
- `GREEN`: Allows vehicles to proceed.

This controls the flow of traffic at intersections.

2.2 Algorithms

This section outlines the primary steps for `traffic_generator` and `simulator` in a clear and structured manner.

2.2.1 traffic_generator Algorithm

1. Initialize a TCP listener on port 55001.

2. Wait for `simulator` to connect and accept the connection.

3. Configure a timer (`spawnClock`) to trigger every 0.8 seconds.

4. When 0.8 seconds have elapsed:

   (a) Randomly select a lane (e.g., `A2`) from 8 possible options. and eligible route (e.g., `MOVE_A_TO_B`) for the chosen lane.

   (b) Create a new vehicle with a unique ID (`nextId++`).

   (c) Position the vehicle off-screen based on the number of existing vehicles in that lane (e.g., 552.5, -20 for the first vehicle in `A2`).

   (d) Package the vehicle's data (ID, lane, route, position) into a packet.

   (e) Send the packet to `simulator` via the socket.

   (f) Reset the timer.

5. Repeat step 4 indefinitely to continuously spawn vehicles.

2.2.2 simulator Algorithm

1. Load the map image and initialize a window with dimensions 1080x1080.

2. Establish a connection to `traffic_generator` on port 55001.

3. Create empty lists for vehicles, lane triggers, and traffic lights.

4. Set up 12 lane triggers and 4 traffic lights at predetermined map positions.

5. Enter the main loop:

   (a) Check for incoming vehicle data from the socket.

   (b) If data is received, create a new vehicle and add it to the list.

   (c) For each vehicle:

   i. If the vehicle collides with `blocker` of a red light, stop it.
   ii. Otherwise, move the vehicle towards its next waypoint.
   iii. Check if the vehicle enters or exits a `LaneTrigger`, updating its lane queue accordingly.
   iv. If the vehicle reaches its destination and completes its route, remove it from the list.

(d) For traffic light:

   i. Calculate the average number of vehicles per road using `Road`.
   ii. Sort roads in `LightQueue` based on traffic (or prioritize `A2` if needed).
   iii. Set one light to `GREEN (for no of avg vehicle x 1 second)` and the others to `RED`.

(e) Draw the map, vehicles, and lights on the window.

6. Continue the loop until the window is closed.

2.3 LaneTrigger

The `LaneTrigger` class detects vehicles within specific lanes.

2.3.1 Attributes

- `position (sf::Vector2f)`: The position of the trigger on the map.
- `size (sf::Vector2f)`: The dimensions of the trigger area.
- `lane (Lane)`: The lane associated with the trigger.
- `laneArea (sf::RectangleShape)`: The visual representation of the trigger.

2.3.2 Methods

- `bool isVehicleOnLane(Vehicle& vehicle)`: Returns true if a vehicle intersects with the trigger area.
- `Lane getLane()`: Returns the lane monitored by the trigger.
- `void draw(sf::RenderWindow& window)`: Renders the trigger on the screen.

2.3.3 Purpose and Approach

I developed this class to track vehicles in designated lanes. The `isVehicleOnLane` method employs collision detection to determine if a vehicle is within the trigger, enabling the system to update lane queues. I set the trigger to be invisible to maintain a clean visual appearance on the map.

2.4 LightQueue

The `LightQueue` class determines the order in which traffic lights turn green based on traffic conditions.

2.4.1 Attributes

- `queue (std::vector<RoadType>)`: A list of roads awaiting a green light.

2.4.2 Methods

- `void enqueue(RoadType road)`: Adds a road to the queue.
- `RoadType dequeue()`: Removes and returns the next road in the queue.
- `void sortQueue(float avgA, float avgB, float avgC, float avgD, int a2Size)`: Sorts roads by traffic volume or prioritizes Road A.
- `bool empty()`: Returns true if the queue is empty.
- `void clear()`: Empties the queue.

2.4.3 Purpose and Approach

I designed this class to enable intelligent traffic light management. The `sortQueue` method prioritizes Road A if Lane A2 has more than 10 vehicles; otherwise, it sorts roads based on the average number of vehicles. I utilized a lambda function to implement flexible sorting logic.

2.5 Road

The `Road` class groups two lanes into a single road entity.

2.5.1 Attributes

- `lane2 (VehicleQueue)`: The queue for the second lane.
- `lane3 (VehicleQueue)`: The queue for the third lane.

2.5.2 Methods

- `float getVehicleAvg(VehicleQueue lane2, VehicleQueue lane3)`: Calculates the average vehicle count across both lanes.

2.5.3 Purpose and Approach

I implemented this class to compute the traffic load on a road. The `getVehicleAvg` method provides an average that informs the traffic light system about which road requires priority.

2.6 TrafficControl

The `TrafficControl` class manages traffic lights and their associated stop zones.

2.6.1 Attributes

- `light (Light)`: The current state (`RED` or `GREEN`).
- `road (RoadType)`: The road controlled by the light.
- `lightShape (sf::CircleShape)`: The visual representation of the light.
- `blocker (sf::RectangleShape)`: The stop zone for vehicles.

2.6.2 Methods

- `void setLight(Light newLight)`: Updates the light's state.
- `Light getLight()`: Returns the current state.
- `RoadType getRoad()`: Returns the associated road.
- `void draw(sf::RenderWindow& window)`: Draws the light and blocker.
- `sf::RectangleShape getBlocker()`: Returns the stop zone.
- `bool isRed()`: Returns true if the light is red.

2.6.3 Purpose and Approach

I created this class to regulate traffic at intersections. The `blocker` ensures vehicles stop at red lights. I addressed the issue of vehicles passing through lights by precisely aligning the blocker with the map layout.

2.7 VehicleClass

The `VehicleClass` defines the behavior and properties of vehicles.

2.7.1 Attributes

- `id (int)`: The unique identifier for the vehicle.
- `origin (Lane)`: The starting lane.
- `destination (Lane)`: The destination lane.
- `route (Route)`: The path to follow.
- `rectangle (sf::RectangleShape)`: The visual shape of the vehicle.
- `waypoints (std::vector<sf::Vector2f>)`: The list of points to traverse.
- `speed (float)`: The vehicle's movement speed.

2.7.2 Methods

- `void update(float deltaTime, const std::vector<Vehicle>& allVehicles)`: Moves the vehicle along its waypoints.
- `void stop()`: Halts the vehicle.
- `void resume()`: Resumes vehicle movement.
- `bool hasCompletedRoute()`: Returns true if the route is complete.
- `void draw(sf::RenderWindow& window)`: Renders the vehicle.
- `sf::FloatRect getBounds()`: Returns the vehicle's bounding box.

2.7.3 Purpose and Approach

I implemented this class to manage vehicle movement. Waypoints guide the vehicle's path, and collision checks in `update` prevent overlaps. I controlled stopping at traffic lights by adjusting the `speed` variable.

2.8 VehicleQueue

The `VehicleQueue` class tracks vehicles waiting in each lane.

2.8.1 Attributes

- `queue (std::vector<Vehicle>)`: The list of vehicles in the lane.

2.8.2 Methods

- `void enqueue(Vehicle& vehicle)`: Adds a vehicle to the queue.
- `void dequeue()`: Removes the first vehicle from the queue.
- `int size()`: Returns the number of vehicles in the queue.

2.8.3 Purpose and Approach

I designed this class to monitor the number of vehicles in each lane, providing essential data for the traffic light system.

2.9 traffic_generator

The `traffic_generator` file generates vehicles and transmits them to `simulator`.

2.9.1 Attributes

- `nextId (int)`: The ID for the next vehicle.
- `spawnInterval (float)`: The time interval between vehicle spawns.
- `spawnClock (sf::Clock)`: Tracks the timing of spawns.

2.9.2 Methods

- `void update(std::vector<Vehicle>& vehicles, sf::TcpSocket& socket, float deltaTime)`: Generates and sends vehicles.

2.9.3 How It Works: Vehicle Lifecycle

The lifecycle of a vehicle begins as follows:

1. **Setup**: The generator listens on port 55001, awaiting a connection from `simulator`.
2. **Spawn Timing**: Every 0.8 seconds (`spawnInterval`), the `update` method executes.
3. **Creation**: A random lane (e.g., `Lane::A2`) and route (e.g., `Route::MOVE_A_TO_B`) are selected. A vehicle (ID 1) is created and positioned off-screen (e.g., 552.5, -20) based on the number of prior vehicles in A2.
4. **Sending**: The vehicle's data (ID, origin, destination, route, position) is packaged into a socket packet and sent to `simulator`.
5. **Repeat**: The process repeats to spawn additional vehicles.

I implemented random selection and spacing to emulate realistic traffic patterns.

2.9.4 Purpose and Approach

This component ensures a steady flow of traffic. I used a gap variable to space vehicles appropriately and learned networking principles to ensure reliable data transmission.

2.10 simulator

The `simulator` file serves as the main driver of the simulation, overseeing all components.

2.10.1 Attributes

- `window (sf::RenderWindow)`: Displays the simulation.
- `queues (std::map<Lane, VehicleQueue>)`: Tracks vehicle queues for each lane.
- `vehicles (std::vector<std::unique_ptr<Vehicle>>)`: Stores all active vehicles.
- `laneTriggers (std::vector<LaneTrigger>)`: Contains lane sensors.
- `trafficControls (std::vector<TrafficControl>)`: Manages traffic lights.
- `lightQueue (LightQueue)`: Determines the order of light changes.
- `a2Priority (bool)`: Prioritizes Road A if Lane A2 has more than 10 vehicles waiting.

2.10.2 How It Works: Vehicle Lifecycle

Consider a vehicle traveling from Lane A2 to Lane B1:

1. **Arrival**: The `simulator` receives data (ID 1, `Lane::A2` to `Lane::B1`, `Route::MOVE_A_TO_B`) from the socket.
2. **Creation**: A `Vehicle` object is created and added to `vehicles`, positioned at the starting point of A2 (e.g., 552.5, -20).
3. **Movement**: Within the main loop, `Vehicle::update` moves the vehicle south along its waypoints. A `LaneTrigger` at A2 detects it, adding it to the A2 queue.
4. **Traffic Light Check**: As the vehicle approaches Road A's light, if `TrafficControl` indicates `RED`, it stops at the `blocker`. The `LightQueue` calculates Road A's average (e.g., 6 vehicles) and may switch

the light to `GREEN` for 6 seconds if it has highest priority according to LightQueue.

5. **Progress**: Once the light turns `GREEN`, the vehicle resumes movement, exits A2's trigger (removing it from the queue), and enters B1's trigger (adding it to B1's queue).

6. **Completion**: The vehicle reaches B1's end. `currentWaypointIndex >= waypoints.size()` returns true, and it is removed from `vehicles`.

I synchronized this process using clocks to adjust light timing dynamically based on traffic.

2.10.3 Purpose and Approach

This file integrates all components of the system. I utilized triggers and queues to monitor traffic and clocks to manage light timing. A critical challenge was ensuring vehicles were continuously updated until route completion, resolved by checking waypoint progress before removal.

Chapter 3: Problem Solving

During development, I encountered several challenges, ranging from basic to complex issues. Below, I detail how I addressed some problems systematically.

3.1 Vehicles Disappearing Prematurely

Vehicles were being removed from the simulation before reaching their destinations. Upon investigation, I found that they were removed when exiting a `LaneTrigger`, rather than at the end of their route. I implemented a check using `currentWaypointIndex >= waypoints.size()`, ensuring vehicles are only removed after completing their waypoints. Thorough testing confirmed that vehicles now persist until their journey is complete.

3.2 Handling Vehicle Collisions

Collisions between vehicles occurred frequently, disrupting the simulation's realism. Initially, vehicles moved without regard for others, leading to overlaps. I researched a distance-checking method and implemented it in `Vehicle::update`. If a vehicle is within 20 units of another, it slows down to maintain spacing. After several adjustments, this approach achieved smoother vehicle movement.

3.3 Avoiding a Tilemap for the Map

I considered using a tilemap to create the road network, but this approach proved overly complex. Managing numerous tiles required extensive coding, which exceeded my current skill level. Instead, I opted to draw a single map image using a graphics tool and loaded it into `simulator` with SFML. This simpler method was effective and met the project's requirements.

3.4 Implementing Vehicle Movement

At first, vehicles remained stationary despite my efforts. I addressed this by defining `waypoints` in `VehicleClass`, representing a list of coordinates for each vehicle to follow. In `update`, I programmed the vehicle to move toward the next waypoint using `speed` and time calculations. After testing various speed values, I settled on 100, which ensured smooth movement across the map.

3.5 Ensuring Vehicles Follow Traffic Lights

Initially, vehicles ignored traffic lights, passing through intersections regardless of the light's state. I introduced a `blocker` in `TrafficControl` and added logic in `simulator` to check if a vehicle intersects with the `blocker` when the light is `RED`. If so, I called `stop`, setting `speed` to 0. When the light turned `GREEN`, I invoked `resume`, restoring `speed` to 100. Extensive testing ensured vehicles adhered to traffic light rules.

Chapter 4: Conclusion

The traffic simulation system successfully models vehicle movement and adaptive traffic lights. Each component fulfills a specific role, and I systematically resolved challenges throughout development. While the system performs well, future enhancements could include varying vehicle speeds or expanding the map to increase complexity.

Chapter 5: Appendix

This chapter provides a comprehensive overview of the classes utilized, along with an analysis of their performance and list of figures.

5.1 Screenshot

- The red lane represent `priority lane A2`, which is given priority when the of the number of waiting vehicles are greater than 10.

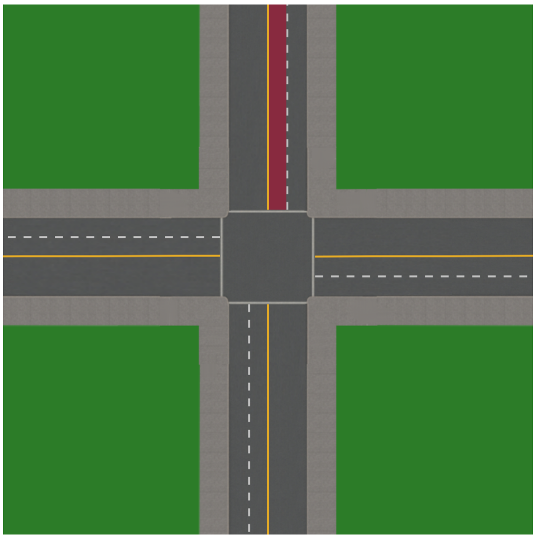

Figure 5.1: Map

5.2 Detailed Screenshot

- The green rectangles represent `laneTriggers`, which are instances of the `LaneTrigger` class.
- The blue rectangle is a traffic blocker and works in accordance with the traffic light. Its position is predetermined by the `TrafficControl` class.
- The red rectangles represent each instance of `Vehicle` class

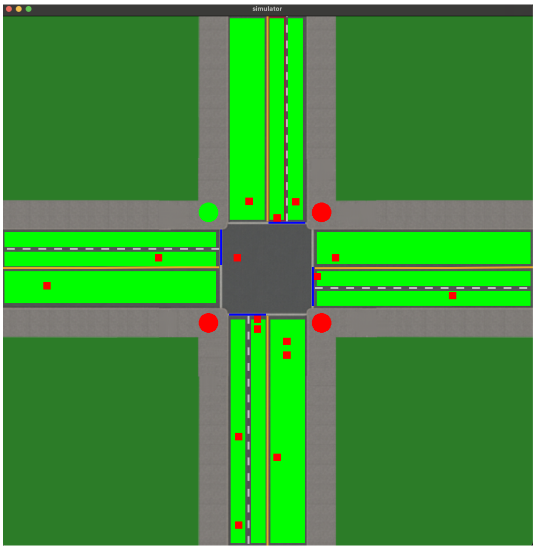

Figure 5.2: Detailed map to explain functionalities during simulation

5.3 Data Structures and Classes Summary

5.3.1 Classes Summary - Part I

Name	How It Works
class LaneTrigger	Uses an sf::RectangleShape to detect vehicle presence through collision checks, adding or removing vehicles from lane queues upon entry or exit.
class LightQueue	Maintains std::vector<RoadType> of roads, sorts them by traffic volume using a lambda function, and dequeues the next road for a green light.
class Road	Manages two VehicleQueue instances, calculating their average vehicle count to inform traffic lights about road congestion.
simulator.cpp	Utilizes a unique pointer of Vehicles to store vehicles, maps lanes to queues with std::map<Lane, VehicleQueue>, and executes the main loop to update and render all elements based on socket data.
traffic_generator.cpp	Maintains std::vector<Vehicle> of spawned vehicles, adds new ones at regular intervals, and transmits their data to simulator via sockets.

Table 5.1: Summary of Classes (Part I)

5.3.2 Classes Summary - Part II

Name	How It Works
`class TrafficControl`	Employs an `sf::CircleShape` for the light's appearance and an `sf::RectangleShape` for the stop zone; updates the light's color and halts vehicles based on its state.
`class VehicleClass`	Stores waypoints in a `std::vector<sf::Vector2f>`, moves vehicles along them, and checks distances to other vehicles to prevent collisions.
`class VehicleQueue`	Keeps a `std::vector<Vehicle>` of vehicles waiting in a lane, adds new vehicles at the end, and removes the first vehicle when it exits the trigger.

Table 5.2: Summary of Classes (Part II)

5.4 Time Complexity Analysis

This section evaluates the time complexity of key operations within the system to assess their efficiency.

Operation	Location	Time Complexity
`isVehicleOnLane`	`LaneTrigger`	O(1)
`sortQueue`	`LightQueue`	O(n log n)
`getVehicleAvg`	`Road`	O(1)
`update` (per vehicle)	`VehicleClass`	O(n)
`enqueue`	`VehicleQueue`	O(1)
`dequeue`	`VehicleQueue`	O(n)
`size`	`VehicleQueue`	O(1)
`main loop` (per frame)	`simulator`	$O(n^2)$
`update` (spawn)	`traffic_generator`	O(n)

Table 5.3: Time Complexity of Key Operations

5.4.1 Explanation

The following explains the time complexities in a clear manner:

- `O(1)`: Indicates constant time complexity, meaning the operation executes quickly regardless of the number of vehicles. For instance, `isVehicleOnLane` performs a simple collision check, which does not scale with the number of elements.

- `O(n)`: Represents linear time complexity, where the operation's duration increases with the number of vehicles. In `VehicleClass::update`, each vehicle checks distances to all other vehicles, resulting in n steps for n vehicles.

- `O(n log n)`: Denotes a slightly higher complexity due to sorting. In `sortQueue`, sorting 4 roads incurs a logarithmic factor, but since n is small, the impact is minimal.

- `O(n²)`: Reflects quadratic time complexity, the least efficient in this system. The `simulator` main loop iterates over each vehicle (n) and checks against all others (n), leading to $n \times n$ operations. This performs adequately with a small number of vehicles but could slow down with larger numbers.

The `O(n²)` complexity suggests potential inefficiencies for large-scale simulations. In the future, I can optimize performance by dividing the map into zones like a tilemap to reduce the number of collision checks.

5.5 GitHub Repository

The source code and video clips for this project are available on GitHub at: `https://github.com/mirajspk/dsa-queue-simulator`.