James M. Mutry

UMGC

**Section 1 – Approach**

This project from the start rattled my nerves. I tried to stay cool and plan the project anyway. My approach was to first set up the GUI, then implement the threading, have the cars adjust speed based on the light states, implement the controls for the buttons and lastly to update the GUI dynamically.

# Section 2 – Assumptions

The assumptions for the project were laid out straightforwardly in the rubric. The Y=0 so there was no need worry about the positioning of that in the cars. I didn’t have to worry about any crazy physics since that was ignored by the rubric. The rubric allowed me to ignore many things that would make this simulation complex.

Section 3 – Not Implemented

Several advanced features were not implemented in this assignment to maintain focus on the core functionalities of the traffic simulation. The simulation does not account for complex road geometries such as turns, intersections, or varying road lengths, as all roads are assumed to be straight. No collision detection or avoidance mechanisms are included, meaning cars do not interact with each other beyond reacting to traffic lights. Realistic acceleration and deceleration physics are omitted, with cars stopping, slowing down, or accelerating instantaneously based on light states. Traffic light synchronization, often used in real-world systems to optimize traffic flow, was not implemented, as each light operates on an independent cycle. Additionally, features such as varying car speeds, pedestrian crossings, and dynamic road conditions (e.g., weather effects) are not included. The visual representation of the simulation is kept minimal, with basic shapes and no detailed animations or vehicle models. These exclusions ensure simplicity and focus while leaving room for future enhancements.

**Section 4 – User Guide**

1. Download the program
2. Unzip the project and place into a folder.
3. Open a new project in preferred java software.
4. Select to start project with the downloaded project that was placed into the folder.

**Section 5 – Lessons Learned**

This assignment provided valuable lessons in designing and implementing a modular, concurrent system using Java. It highlighted the importance of thread synchronization and effective concurrency management, as multiple threads (for cars, traffic lights, and the clock) needed to operate independently while maintaining cohesive simulation behavior. The project reinforced the significance of a well-structured GUI, ensuring that components like the TrafficPanel and InfoPanel dynamically updated to reflect real-time changes without lag or flicker. It also emphasized the importance of clear assumptions and simplified models when tackling complex real-world problems, enabling incremental development and testing. **Section 6 – Possible Improvements**

Introducing more realistic physics, such as gradual acceleration and deceleration for cars, would make the simulation more lifelike. Adding features like collision detection and dynamic interaction between cars could simulate traffic congestion and real-world scenarios. Synchronizing traffic lights to optimize traffic flow would improve realism and demonstrate more advanced traffic management strategies. Expanding the road layout to include intersections, turns, and multiple lanes would add complexity and provide a richer simulation environment.

**Section 7 – UML**

**A diagram of a computer

Description automatically generated**

**Section 8 – Source Code**

import javax.swing.\*;  
import java.awt.\*;  
  
public class MainFrame extends JFrame {  
 private JButton startButton, pauseButton, stopButton, addCarButton;  
 private TrafficPanel trafficPanel; // Make sure it's declared as TrafficPanel  
 private JPanel controlPanel;  
 private InfoPanel infoPanel;  
 private SimulationManager simulationManager;  
  
 public MainFrame() {  
 setTitle("Traffic Simulation");  
 setSize(1200, 800); // Increased window size  
 setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
  
 // Initialize components  
 simulationManager = new SimulationManager();  
 trafficPanel = new TrafficPanel(simulationManager); // This is TrafficPanel  
 controlPanel = new JPanel();  
 infoPanel = new InfoPanel(simulationManager);  
  
 // Create buttons  
 startButton = new JButton("Start");  
 pauseButton = new JButton("Pause");  
 stopButton = new JButton("Stop");  
 addCarButton = new JButton("Add Car");  
  
 // Add button functionality  
 startButton.addActionListener(e -> simulationManager.startSimulation());  
 pauseButton.addActionListener(e -> simulationManager.pauseSimulation());  
 stopButton.addActionListener(e -> simulationManager.stopSimulation());  
 addCarButton.addActionListener(e -> simulationManager.addCar());  
  
 // Organize layout  
 controlPanel.add(startButton);  
 controlPanel.add(pauseButton);  
 controlPanel.add(stopButton);  
 controlPanel.add(addCarButton);  
  
 setLayout(new BorderLayout());  
 add(trafficPanel, BorderLayout.*CENTER*);  
 add(controlPanel, BorderLayout.*SOUTH*);  
 add(infoPanel, BorderLayout.*EAST*);  
  
 // Periodic refresh for the TrafficPanel  
 Timer timer = new Timer(100, e -> trafficPanel.refresh()); // Call refresh on TrafficPanel  
 timer.start();  
 }  
  
 public static void main(String[] args) {  
 SwingUtilities.*invokeLater*(() -> {  
 MainFrame frame = new MainFrame();  
 frame.setVisible(true);  
 });  
 }  
}

public class Car extends Thread {  
 private String name;  
 private int speed; // m/s  
 private int position;  
 private boolean running;  
 private boolean paused;  
  
 public Car(String name, int speed) {  
 this.name = name;  
 this.speed = speed;  
 this.position = 0;  
 running = true;  
 paused = false;  
 }  
  
 public void run() {  
 while (running) {  
 if (!paused) {  
 position += speed; // Update position  
 System.*out*.println(name + " Position: " + position + " meters");  
 try {  
 Thread.*sleep*(1000); // 1-second interval  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 public void pause() {  
 paused = true;  
 }  
  
 public void stopThread() {  
 running = false;  
 }  
}

public class CarThread extends Thread {  
 private String carName;  
 private int speed; // Current speed in meters/second  
 private int position; // Current position in meters  
 private TrafficLightThread trafficLight; // Associated traffic light  
 private boolean running;  
 private boolean paused;  
  
 public CarThread(String carName, TrafficLightThread trafficLight) {  
 this.carName = carName;  
 this.speed = 10; // Default speed  
 this.position = 0;  
 this.trafficLight = trafficLight;  
 this.running = true;  
 this.paused = false;  
 }  
  
 @Override  
 public void run() {  
 while (running) {  
 try {  
 if (!paused) {  
 // Adjust speed based on traffic light state  
 String lightState = trafficLight.getLightState(); // Fetch the light state  
 switch (lightState) {  
 case "Green":  
 speed = 20; // Speed up  
 break;  
 case "Yellow":  
 speed = 5; // Slow down  
 break;  
 case "Red":  
 speed = 0; // Stop completely  
 break;  
 }  
  
 // Update position only if speed > 0  
 if (speed > 0) {  
 position += speed;  
 System.*out*.println(carName + " is at position " + position + " meters, speed: " + speed + " m/s");  
 } else {  
 System.*out*.println(carName + " is stopped at position " + position + " meters due to red light.");  
 }  
  
 // Simulate one-second intervals  
 Thread.*sleep*(1000);  
 } else {  
 Thread.*sleep*(100); // Shorter sleep while paused  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public int getPosition() {  
 return position;  
 }  
  
 public int getSpeed() {  
 return speed;  
 }  
  
 public String getCarName() {  
 return carName;  
 }  
  
 public void pause() {  
 this.paused = true;  
 }  
  
 public void resumeCar() {  
 this.paused = false;  
 }  
  
 public void stopThread() {  
 this.running = false;  
 }  
}

public class ClockThread extends Thread {  
 private int currentTime; // Tracks the simulation time in seconds  
 private boolean running;  
 private boolean paused;  
  
 public ClockThread() {  
 this.currentTime = 0; // Start at 0 seconds  
 this.running = true;  
 this.paused = false;  
 }  
  
 @Override  
 public void run() {  
 while (running) {  
 try {  
 if (!paused) {  
 System.*out*.println("Simulation Time: " + currentTime + " seconds");  
 currentTime++; // Increment time by 1 second  
 Thread.*sleep*(1000); // 1-second interval  
 } else {  
 Thread.*sleep*(100); // Shorter sleep while paused  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public int getCurrentTime() {  
 return currentTime;  
 }  
  
 public void pause() {  
 paused = true;  
 }  
  
 public void resumeClock() {  
 paused = false;  
 }  
  
 public void stopThread() {  
 running = false;  
 }  
}

import javax.swing.\*;  
import java.awt.\*;  
  
public class InfoPanel extends JPanel {  
 private SimulationManager simulationManager;  
 private JLabel clockLabel;  
 private JTextArea carInfoArea;  
  
 public InfoPanel(SimulationManager simulationManager) {  
 this.simulationManager = simulationManager;  
 setLayout(new BorderLayout());  
 setPreferredSize(new Dimension(300, 600));  
  
 // Create components  
 clockLabel = new JLabel("Simulation Time: 0 seconds");  
 carInfoArea = new JTextArea();  
 carInfoArea.setEditable(false);  
  
 // Add components to the panel  
 add(clockLabel, BorderLayout.*NORTH*);  
 add(new JScrollPane(carInfoArea), BorderLayout.*CENTER*);  
  
 setBackground(Color.*LIGHT\_GRAY*);  
 setBorder(BorderFactory.*createTitledBorder*("Simulation Info"));  
  
 // Start updating the panel dynamically  
 new Timer(1000, e -> updateInfo()).start();  
 }  
  
 private void updateInfo() {  
 // Fetch the current time from the ClockThread  
 int currentTime = simulationManager.getClockThread().getCurrentTime();  
 clockLabel.setText("Simulation Time: " + currentTime + " seconds");  
  
 // Update car information  
 StringBuilder carInfo = new StringBuilder();  
 for (CarThread car : simulationManager.getCars()) {  
 carInfo.append(car.getCarName())  
 .append(" - Position: ").append(car.getPosition())  
 .append(" m, Speed: ").append(car.getSpeed()).append(" m/s\n");  
 }  
 carInfoArea.setText(carInfo.toString());  
 }  
}

import java.util.ArrayList;  
import java.util.List;  
  
public class SimulationManager {  
 private ClockThread clockThread;  
 private List<TrafficLightThread> trafficLights;  
 private List<CarThread> cars;  
 private boolean running;  
  
 public ClockThread getClockThread() {  
 return clockThread;  
 }  
 public SimulationManager() {  
 clockThread = new ClockThread();  
 trafficLights = new ArrayList<>();  
 cars = new ArrayList<>();  
 running = false;  
  
 // Initialize three traffic lights  
 for (int i = 0; i < 3; i++) {  
 trafficLights.add(new TrafficLightThread("Intersection " + (i + 1)));  
 }  
 }  
  
 public void startSimulation() {  
 if (!running) {  
 running = true;  
 clockThread.start();  
 trafficLights.forEach(Thread::start);  
 cars.forEach(Thread::start);  
 }  
 }  
  
 public void pauseSimulation() {  
 running = false;  
 clockThread.pause();  
 trafficLights.forEach(TrafficLightThread::pause);  
 cars.forEach(CarThread::pause);  
 }  
  
 public void stopSimulation() {  
 running = false;  
 clockThread.stopThread();  
 trafficLights.forEach(TrafficLightThread::stopThread);  
 cars.forEach(CarThread::stopThread);  
 }  
  
 public void addCar() {  
 int trafficLightIndex = cars.size() % trafficLights.size(); // Cycle through traffic lights  
 TrafficLightThread assignedLight = trafficLights.get(trafficLightIndex);  
 CarThread car = new CarThread("Car " + (cars.size() + 1), assignedLight);  
 cars.add(car);  
 if (running) {  
 car.start();  
 }  
 }  
  
 public List<CarThread> getCars() {  
 return cars;  
 }  
  
 public List<TrafficLightThread> getTrafficLights() {  
 return trafficLights;  
 }  
}

import javax.swing.\*;  
import java.awt.\*;  
import java.util.List;  
  
public class TrafficPanel extends JPanel {  
 private SimulationManager simulationManager;  
  
 public TrafficPanel(SimulationManager simulationManager) {  
 this.simulationManager = simulationManager;  
 setPreferredSize(new Dimension(1000, 600)); // Ensure panel is large enough  
 setBackground(Color.*WHITE*);  
 }  
  
 @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
  
 // Draw traffic lights  
 List<TrafficLightThread> trafficLights = simulationManager.getTrafficLights();  
 for (int i = 0; i < trafficLights.size(); i++) {  
 String lightState = trafficLights.get(i).getLightState(); // Fetch light state  
 int x = 50 + i \* 300; // Space intersections evenly  
 int y = 50;  
  
 // Draw intersection block  
 g.setColor(Color.*BLACK*);  
 g.drawRect(x, y, 50, 150);  
 g.drawString("Intersection " + (i + 1), x - 10, y - 10);  
  
 // Draw light state as a colored circle  
 switch (lightState) {  
 case "Red":  
 g.setColor(Color.*RED*);  
 break;  
 case "Yellow":  
 g.setColor(Color.*YELLOW*);  
 break;  
 case "Green":  
 g.setColor(Color.*GREEN*);  
 break;  
 }  
 g.fillOval(x + 10, y + 10, 30, 30); // Circle representing the light  
 }  
  
 // Draw cars  
 List<CarThread> cars = simulationManager.getCars();  
 for (int i = 0; i < cars.size(); i++) {  
 int carPosition = cars.get(i).getPosition();  
 int x = carPosition % getWidth(); // Wrap around the panel width  
 int y = 300 + i \* 50; // Space cars vertically  
  
 // Ensure car stays within bounds  
 if (carPosition >= 0) {  
 // Draw car  
 g.setColor(Color.*BLUE*);  
 g.fillRect(x, y, 50, 20);  
 g.setColor(Color.*BLACK*);  
 g.drawString(cars.get(i).getCarName(), x, y - 5);  
 }  
 }  
 }  
  
 // Repaint periodically to reflect simulation updates  
 public void refresh() {  
 repaint();  
 }  
}

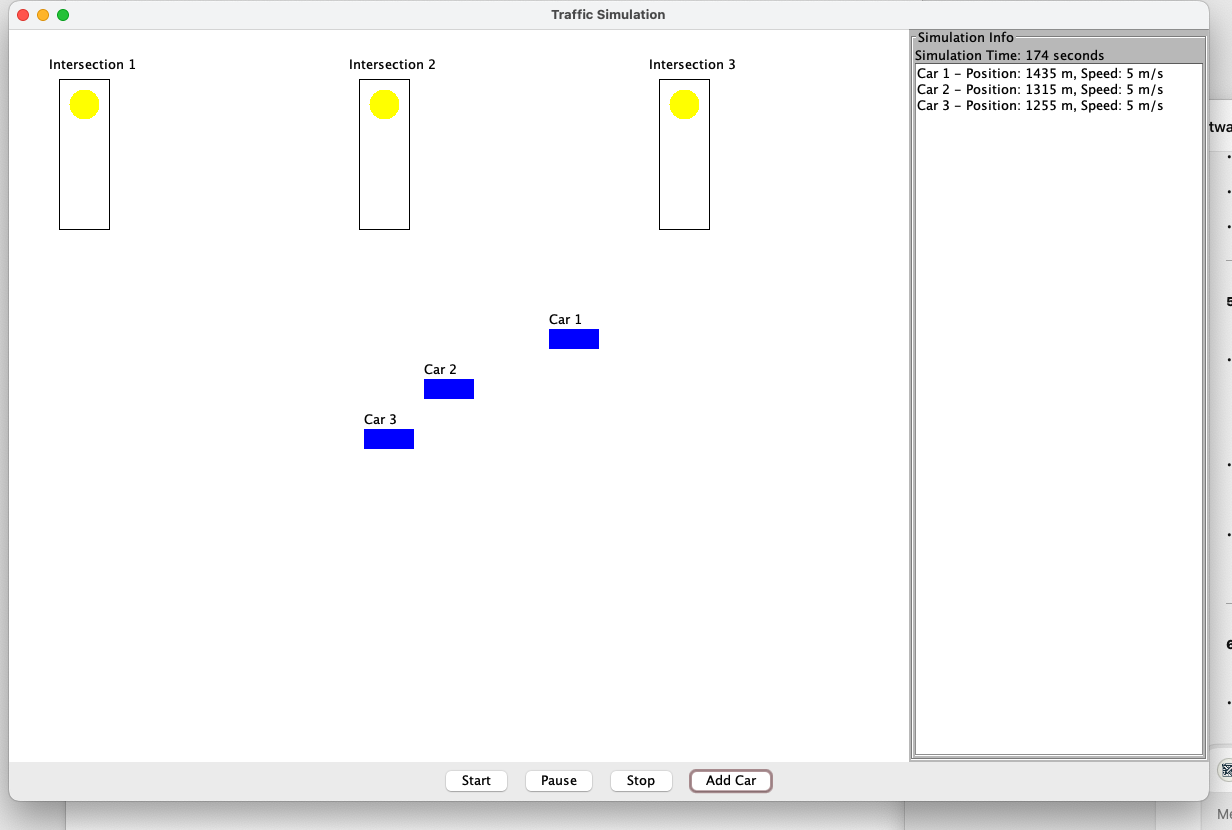
**Section 9 – Test Plans**

****

**Screenshots**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**