MAJOR PROJECT: PR1107

PROJECT REPORT

TOPIC:

SIMULATION OF AUTONOMOUS VEHICLE MOVEMENT USING PYTHON, PY-GAME, AND PATHFINDING ALGORITHMS

BY:

TIRUMALA SUMITH (2021BTECH114)

ASIT KUMAR GIRI (2021BTECH028)

INDEX

1. Abstract

This initiative creates an all-encompassing simulation structure for Autonomous Vehicle (AV) navigation, incorporating two essential levels of autonomy: Local Collision Avoidance and Global Path Planning. Although conventional simulations frequently separate these elements, this system integrates them to showcase a comprehensive decision-making framework from immediate safety to long-term routing.

At the micro-level (Local Planner), the system uses a detailed grid environment where the AV implements a predictive occupancy model to identify and avoid changing hazards. This module addresses urgent dangers, like shifting obstacle vehicles and stationary potholes, guaranteeing physical safety through responsive driving

At the macro level (Global Planner), the simulation extends to actual urban networks through OpenStreetMap (OSMnx). It utilizes a Modified Dijkstra Algorithm that actively determines optimal paths through intricate city networks (e.g., Jaipur, London) by incorporating real-time Traffic density penalties instead of solely considering physical distance.

The outcome is a dual-mode simulation that confirms the AV's capability to devise efficient routes across cities while ensuring immediate physical safety, illustrated through synchronized occupancy matrices, schematic graphs, and real-time path telemetry

2. Introduction

2.1 Motivation

Self-driving cars (SDCs) signify the next era of smart transport, offering the potential to decrease accidents and enhance traffic efficiency. Nonetheless, building a dependable AV involves addressing two fundamentally distinct issues at the same time: Safety and Efficiency.

A vehicle needs to respond immediately to urgent dangers—like a pedestrian emerging or a car changing lanes—while also strategizing a long-term path through a complicated city grid that considers traffic congestion and road obstructions. Standard simulations typically concentrate on just one of these elements: either basic grid-based obstacle evasion or theoretical graph-based navigation. This initiative seeks to close that divide by creating a thorough simulation environment that merges local collision avoidance with global traffic-aware navigation, offering a complete perspective on AV decision-making processes.

2.2 Project Objectives

The main goal of this project is to create a strong navigation framework that functions at two different levels of abstraction.

The objectives are:

- To Build a Micro-Navigation System: Design a detailed grid simulation that enables the AV to utilize predictive algorithms for identifying and avoiding moving obstacles (cars) and immovable dange rs (potholes) instantly
- For Global Path Planning Implementation: Use OpenStreetMap (OSMnx) to map actual city networks (e.g., Jaipur, London) into the simulation and apply a Modified Dijkstra Algorithm that emphasizes travel time rather than distance by imposing penalties on congested roads.
- To Integrate Hierarchical Control: Show how overarching routing choices (Macro) convert into localized move ment actions (Micro) while maintaining safety.
- To Illustrate Decision Logic: Deliver detailed telemetry via dual-view dashboards, featuring Schematic Overviews for route planning and Occupancy Matrices for collision assessment

2.3 Scope

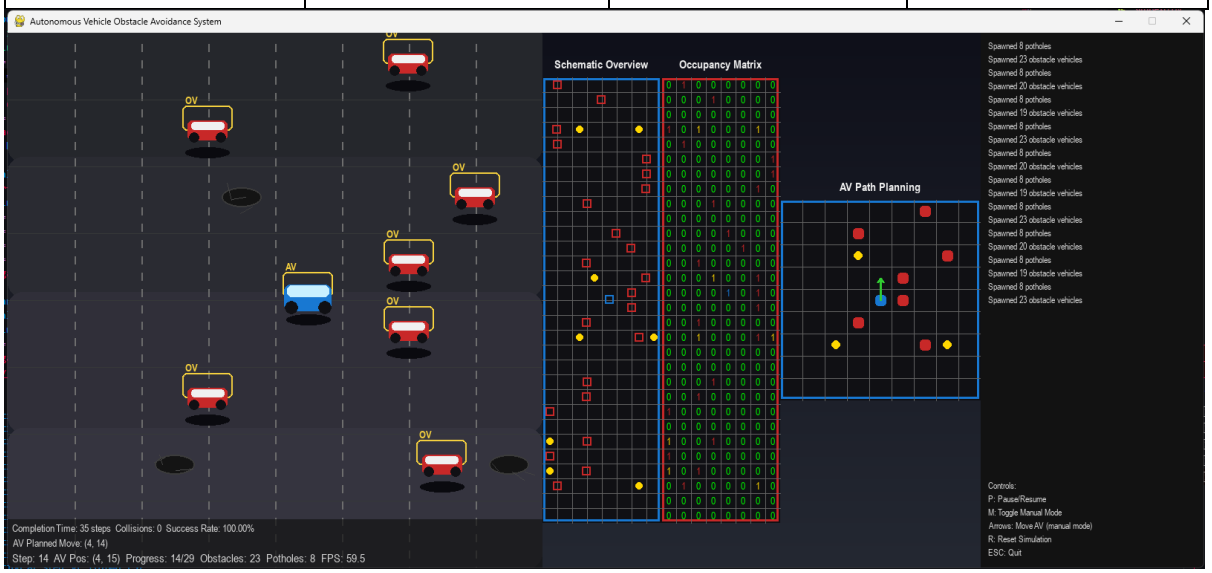
The scope of this system encompasses two synchronized simulation environments:

1. **The Local Environment (Micro-Scope):**
    - Simulates an 8×30 road grid.
    - Focuses on **Reactive Safety**: The AV monitors immediate surroundings to prevent physical crashes.
    - Handles dynamic obstacles that move laterally and static road defects.
2. **The Global Environment (Macro-Scope):**
    - Simulates city-scale graph networks with thousands of nodes.
    - Focuses on **Predictive Efficiency**: The AV calculates routes based on traffic density data gathered from simulated "Traffic Bots."
    - Handles complex decision-making, such as re-routing when a user manually blocks a road or when traffic density exceeds a critical threshold.

3. System Architecture

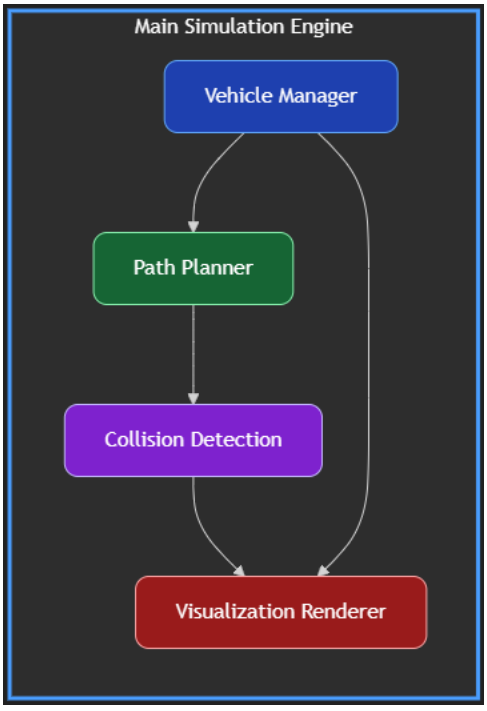
3.1 Overall Design

MAIN SIMULATION ENGINE			
Vehicle Manager	Path Planer	Collision Detection	Visualization Renderer
• Spawns Av and OV's. • Tracks positions. • Manages animation.	• Predicts future states. • Uses will_be_free(). • Prioritizes forward move.	• Checks occupancy conflicts. • Triggers Crash/Success. • Logs events.	• Renders road, vehicles, potholes. • Updates HUD & Panels.



3.2 Key Configuration Parameters

Parameter	Value	Description
GRID_COLS	8	Number of columns in the grid.
GRID_ROWS_ONSCREEN	8	Visible rows on screen.
TOTAL_ROAD_ROWS	30	Total Length of the road.
CELL	90 pixels	Size of each grid cell.
FPS	60	Frames per second
STEP_SECONDS	0.5	Time between decision steps
SPAWN_PROB	0.68	Obstacle spawn probability
POTHOLE_PROB	0.15	Pothole spay probability



3.3 Software Stack

- **Language:** Python 3.x
- **Graphics Library:** Pygame
- **Data Structures:** Dataclasses, Sets, Dictionaries
- **Animation:** Custom easing functions

1. Main Simulation Engine

This serves as the core of the program, encompassing the primary while loop. It orchestrates all other modules, oversees the overall state of the simulation (such as paused or manual_mode), and regulates the timing and update ticks of the game.

2. Vehicle Manager

This element oversees the status of all vehicles. It is tasked with spawning, deleting, and resetting the AV and OVs. Additionally, it updates their animation frames (anim_frame) to visually transition them from their current cell to their target cell during each frame.

3. Path Planner

This functions as the intellect of the simulation. It contains the logic for decision-making. For the AV, it executes the av_decide_and_move algorithm to ascertain the safest subsequent step. For obstacle vehicles, it implements plan_obstacle_moves to facilitate their movement and generate dynamic hazards.

4. Collision Detection

This module serves as the observer and adjudicator. It undertakes two primary functions:

Proactive: It offers the will_be_free function, enabling the Path Planner to verify if a prospective move is secure from potholes or other vehicles.

Reactive: Following the execution of moves, it assesses whether the AV's final position is atop an OV or pothole, indicating a "CRASH" state to the Main Engine.

5. Visualization Renderer

This represents the graphics engine. It lacks logic; it solely renders what it is instructed to. This encompasses all draw_... functions for vehicles, potholes, and the road. Furthermore, it visualizes all UI/HUD components such as the schematic overview, occupancy matrix, and log panel.

4. Tools and Dependencies

To replicate the dual-layer simulation environment (Grid + City Map), specific software tools and Python libraries were utilized. The project relies on the Python ecosystem due to its robust support for both graphical simulations and complex graph-theory calculations.

Core Environment

- **Language:** Python 3.9+
    - *Reasoning:* Python was chosen for its extensive library support (OSMnx, NetworkX) which allows for rapid integration of real-world geospatial data into the simulation logic.
- **IDE:** VS Code / PyCharm
    - *Usage:* Used for modular code development, debugging the simulation loop, and managing the virtual environment.

Key Libraries and Modules

The simulation requires the following external packages to function:

1. **Pygame (pygame)**
    - **Role:** The core rendering engine.
    - **Usage:** It handles the 60 FPS simulation loop, draws the 2D grid, renders vehicle sprites, and manages user inputs (keyboard/mouse). It is the backbone of the **Micro-Planner** visualization.
2. **OSMnx (osmnx)**
    - **Role:** The map data provider.
    - **Usage:** It connects to the OpenStreetMap API to download real-world street networks (e.g., Jaipur, London). It converts raw geospatial data (Latitude/Longitude) into graph projections suitable for navigation.
3. **NetworkX (networkx)**
    - **Role:** The mathematical graph solver.
    - **Usage:** It stores the city map as a mathematical structure (Nodes and Edges). It is responsible for executing the **Modified Dijkstra Algorithm** to calculate optimal paths based on the traffic-density weights provided by the simulation.
4. **Pygame-Menu (pygame_menu)**
    - **Role:** UI/UX Interface.
    - **Usage:** It powers the initial configuration screen, allowing users to select cities, set pothole density, and choose visual themes before the simulation starts.
5. **Matplotlib (matplotlib)**
    - **Role:** Data Visualization.
    - **Usage:** Used to generate static graphs for the "Dashboard" panel, visualizing success rates, collision counts, and completion times over multiple runs.

3.3 Installation Procedure

To set up the development environment, a virtual environment is recommended to prevent conflict with global system packages.

Step 1: System Prerequisites Ensure Python 3.x and pip are installed on the system.

Step 2: Dependency Installation The following command installs all required libraries:

Bash:

"pip install pygame pygame-menu osmnx networkx matplotlib requests"

Step 3: Verifying OSMnx Configuration Since OSMnx relies on geopandas and shapely, Windows users may need to install pre-compiled binary wheels (e.g., Rtree, GDAL) if the direct pip install fails. On Linux/MacOS, the standard command is sufficient.

5. Path Planning Algorithm

The path-finding system in our project operates on top of a **real-world road network**, which is downloaded using **OSMnx** and represented as a projected NetworkX graph. Unlike the grid-based obstacle simulation from the mid-term, the end-term system uses a *true city map* and performs navigation using dynamically computed shortest paths.

The goal of the algorithm is to allow the Autonomous Vehicle (AV) to:

1. Navigate from one manually selected waypoint to the next
2. Adapt its route if traffic density increases
3. Avoid user-blocked edges
4. Recalculate the shortest path in real time

The complete routing flow integrates **graph abstraction**, **multi-waypoint sequencing**, and **dynamic congestion-aware cost updates**.

---

1. Abstract Graph Generation

Real city road networks contain thousands of nodes, which makes direct visualization and interaction difficult.
To simplify this, we generate an **Abstract Graph**:

- We select high-degree important intersections
- Ensure minimum spacing (250+ pixels apart)
- Assign alphabetical labels (A, B, C, …)
- Compute inter-node connectivity using shortest paths

This graph becomes the **user interface for selecting path waypoints**.

Each abstract node corresponds to a real position in the OSM road graph.

---

2. Waypoint-Based Navigation

The user can interactively select nodes by **right-clicking**, building a list of ordered waypoints:

[Start → Intermediate → … → Destination]

The AV then travels segment-by-segment:

- Segment 1: Waypoint[0] → Waypoint[1]
- Segment 2: Waypoint[1] → Waypoint[2]
- …
- Final segment: Waypoint[n-2] → Waypoint[n-1]

At the end of each segment, the vehicle pauses briefly ("Scanning…") before planning the next leg.

This mimics **real autonomous navigation cycles** where a vehicle makes decisions section-wise.

---

3. Traffic-Aware Path Cost Adjustment

Each road segment (edge) in the OSM graph stores an attribute:

length (in meters)

During simulation, dozens of **traffic bots** move randomly along the map.
Their density is used to calculate congestion on each edge:

traffic_density = number of bots currently moving across the edge

Before the AV calculates a path, the system updates each edge's cost:

current_cost = length × (1 + traffic_density² × 0.1)

This achieves:

✓ **Realistic congestion modeling**

✓ **Exponential penalty for traffic jams**

✓ **Encourages rerouting through clearer roads**

Thus the AV always chooses the most optimal path considering **distance + traffic**.

---

4. Shortest Path Computation (Modified Dijkstra)

The route between two waypoints is calculated using:

nx.shortest_path(G, source, target, weight="current_cost")

This ensures:

- Low traffic routes are preferred
- User-blocked edges are ignored
- Dynamic changes are reflected instantly

If a path becomes unavailable (fully blocked), the AV enters:

STATE = "BLOCKED"

ALERT = "Path Blocked!"

---

5. Movement Execution Along the Path

Once the shortest path is computed:

1. The path is stored as a list of nodes
2. The AV interpolates smoothly between them
3. The sprite rotates toward its direction of travel
4. Each node-to-node movement respects the simulation speed slider

When the AV reaches the last node of the segment:

- It enters a WAITING state (2 seconds)
- Then computes the next segment of the route

This gives a natural, realistic animation rather than teleporting node jumps.

---

6. Handling User-Blocked Roads

Users may press **B** to toggle a blocked road edge.

- If an edge is blocked → AV avoids it
- If blocking disconnects the road → AV raises warning
- The path recalculates automatically on next leg

This demonstrates robust handling of **dynamic road closures**, similar to GPS navigation systems.

---

7. Visual Path Representation

The algorithm's output is visualized in three ways:

Active Path (Cyan Glow)

Highlighted in real-time on the map as the AV is following it.

Future Path (Yellow)

Planned legs between future waypoints.

Abstract Graph Visualization

Shows:

- Nodes (A–L)
- Edges
- Distance labels
- Current segment (highlighted)

This dual-view representation allows users to understand both:

- The macro path (graph view)
- The micro path (actual OSM roads)

8.general Algorithm

The logic for autonomous navigation functions within a hierarchical loop. The Global Planner operates as needed (when a destination is determined or a road is obstructed), whereas the Local Planner functions persistently (during each simulation tick).

Algorithm 1: Global Path Planning (Modified Dijkstra)

Input: Graph $G(V, E)$, Start Node S, Destination Node D, Traffic Density Map T.

Output: Optimal sequence of nodes $P = [n_1, n_2, \dots, n_k]$.

1. **Initialize:**
    - Set distance to $S = 0$, all other nodes $= \infty$.
    - Priority Queue $Q \leftarrow [(0, S)]$.
    - Predecessor Map $\pi \leftarrow \emptyset$.
2. **While Q is not empty:**
    - Extract node u with the minimum cost from Q.
    - **If** $u == D$: **Return** reconstructed path from π.
    - **For each** neighbor v of u connected by edge e:
        - **Get Traffic Data:** Retrieve count of bots t on edge e.
        - Calculate Dynamic Cost:

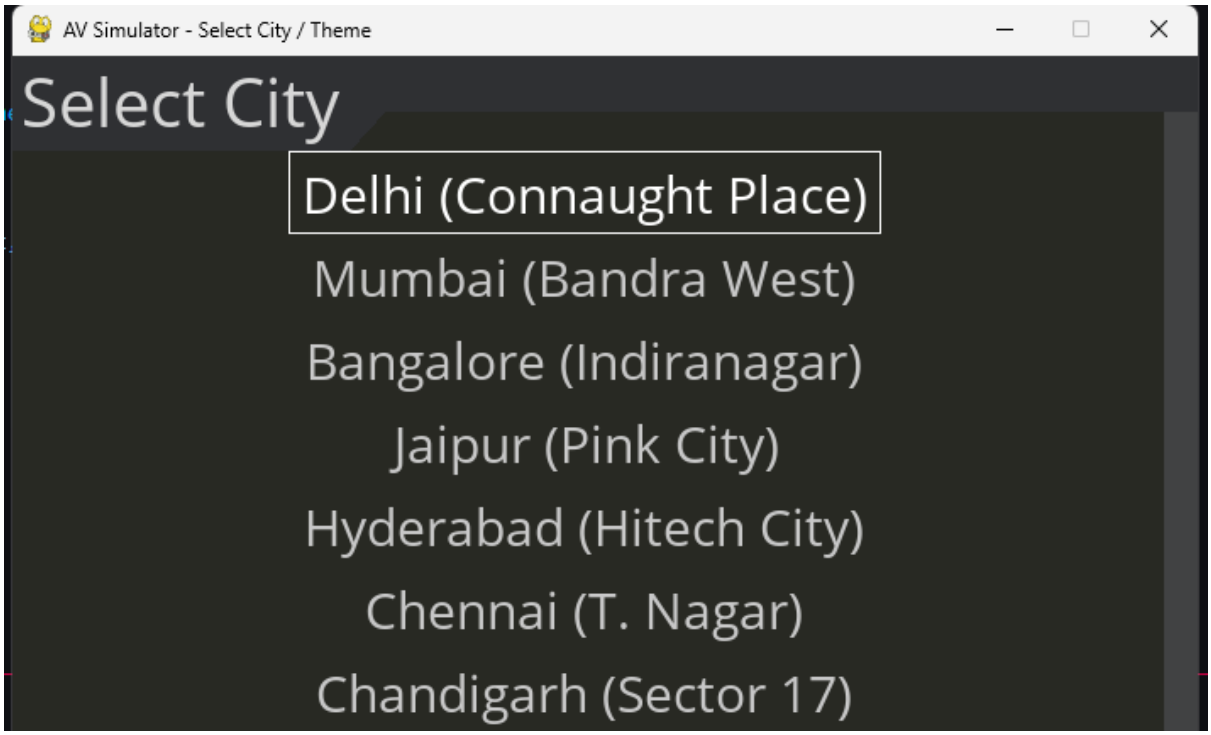
$$W_{new} = Length(e) \times (1 + t^2 \times 0.1)$$

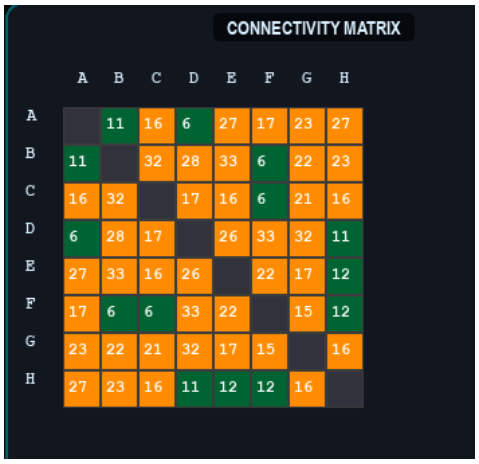
- **Relaxation:**

- **If** $Dist[u] + W_{new} < Dist[v]$:
    - $Dist[v] \leftarrow Dist[u] + W_{new}$
    - $\pi[v] \leftarrow u$
    - Push $(Dist[v], v)$ to Q.

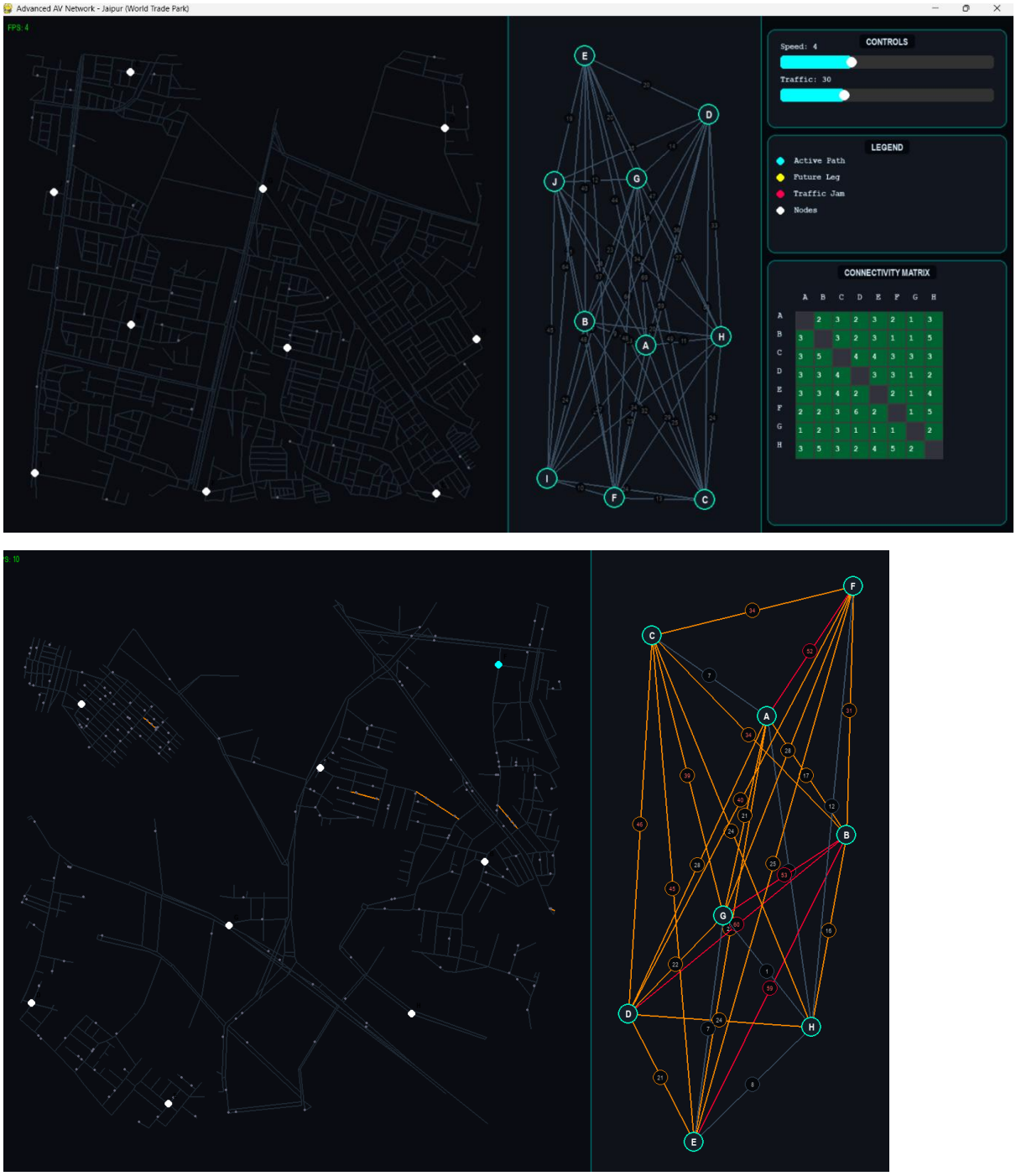
3. Error Handling:

- **If** Q implies empty and D not reached: Trigger "Path Blocked" state.





- Adjacency matrix shows the weight /traffic in the paths from one node to another.



- When there is a high traffic the graph view shows the weight indication .

6.Collision Detection

Collision detection plays a critical role in ensuring safe navigation of the Autonomous Vehicle (AV) within the simulated dynamic grid environment. In this simulation, collisions may occur between:

AV and Obstacle Vehicles (OVs)

AV and Potholes

OVs and Potholes (environmental hazards)

The system must detect all such events both before a move (predictive check) and after a move (terminal check).
Base.py implements a two-layer collision detection mechanism that prevents unsafe movement and accurately signals failures.

---

1. Environment Representation for Collision Detection

The entire road environment is represented as an:

8 × 30 grid

Each cell can be:

AV — the autonomous vehicle

OV — an obstacle vehicle

P — pothole (static hazard)

0 — empty/free space

This grid is constantly updated after every simulation tick, which allows for real-time collision analysis.

---

2. Predictive (Pre-Move) Collision Checking

Before the AV or any obstacle vehicle is allowed to move, the algorithm verifies if the destination cell will be safe.
This is implemented through the key function:

will_be_free(cell)

It checks:

✓ Boundary constraints

Cell must be inside the grid.

✓ Pothole presence

A pothole makes a cell permanently unsafe.

✓ Moving obstacle vehicles

If an obstacle is currently occupying a cell, the system predicts whether:

It will vacate that cell, or

Another obstacle is about to enter it

✓ Simultaneous arrival conflicts

Two vehicles attempting to move into the same cell are resolved using the planned targets map.

This predictive step ensures the AV never chooses a move that will result in a collision in the next frame, even if the hazard is not present yet.

---

3. Post-Move Collision Detection (Terminal Check)

After all move decisions for the current step are committed and the animation completes, the system performs a terminal safety check:

Case 1 — AV enters a pothole

If AV.pos in potholes
→ Immediate CRASH
→ Simulation displays a collision overlay
→ System resets after a short delay

Case 2 — AV collides with an obstacle vehicle

If the AV's final cell equals the position of any obstacle vehicle
→ CRASH event triggered

Case 3 — Obstacle moves into a pothole

Although the AV avoids potholes, obstacle vehicles may enter potholes.
This event is visually depicted but does not end the simulation.

These checks guarantee that every update step ends with a validated, collision-free configuration unless an unavoidable crash has occurred.

---

4. Collision Logic for Obstacle Vehicles

Obstacle vehicles move horizontally (left or right).
The algorithm ensures they:

Do not move outside grid boundaries

Reverse direction when encountering edges or blocked cells

Do not move into potholes

Do not move into other vehicles

This prevents OV–OV collisions and maintains a realistic traffic flow.

---

5. Collision Response Mechanism

When a collision is detected, the simulation enters an overlay state, either:

"CRASH" — unsafe event

"SUCCESS" — AV reached the end safely

In CRASH mode:

Simulation freezes current frame

Displays a dedicated crash overlay

Records metrics (crash count)

Automatically restarts simulation after a delay

This creates a clean and understandable visual response for users.

---

6. Occupancy Matrix as a Collision Visualisation Tool

The Occupancy Matrix Panel provides a binary 8×30 matrix where:

1 indicates an occupied cell

0 indicates a free cell

It visually aids collision understanding by showing:

AV's cell

All OVs

Potholes

This helps users analyze why certain moves were allowed or blocked by the algorithm.

---

7. Summary of Collision Detection Workflow

Pre-Move (Predictive):

Predict where each obstacle will move

Predict whether AV’s candidate moves will become occupied

Reject moves that lead to conflict

Post-Move (Terminal):

Check AV final position

Collision with OV → crash

Collision with pothole → crash

Update environment state

Visual Feedback:

1. Red crash overlay
2. Updated logs
3. Metrics counter

General Algorithm

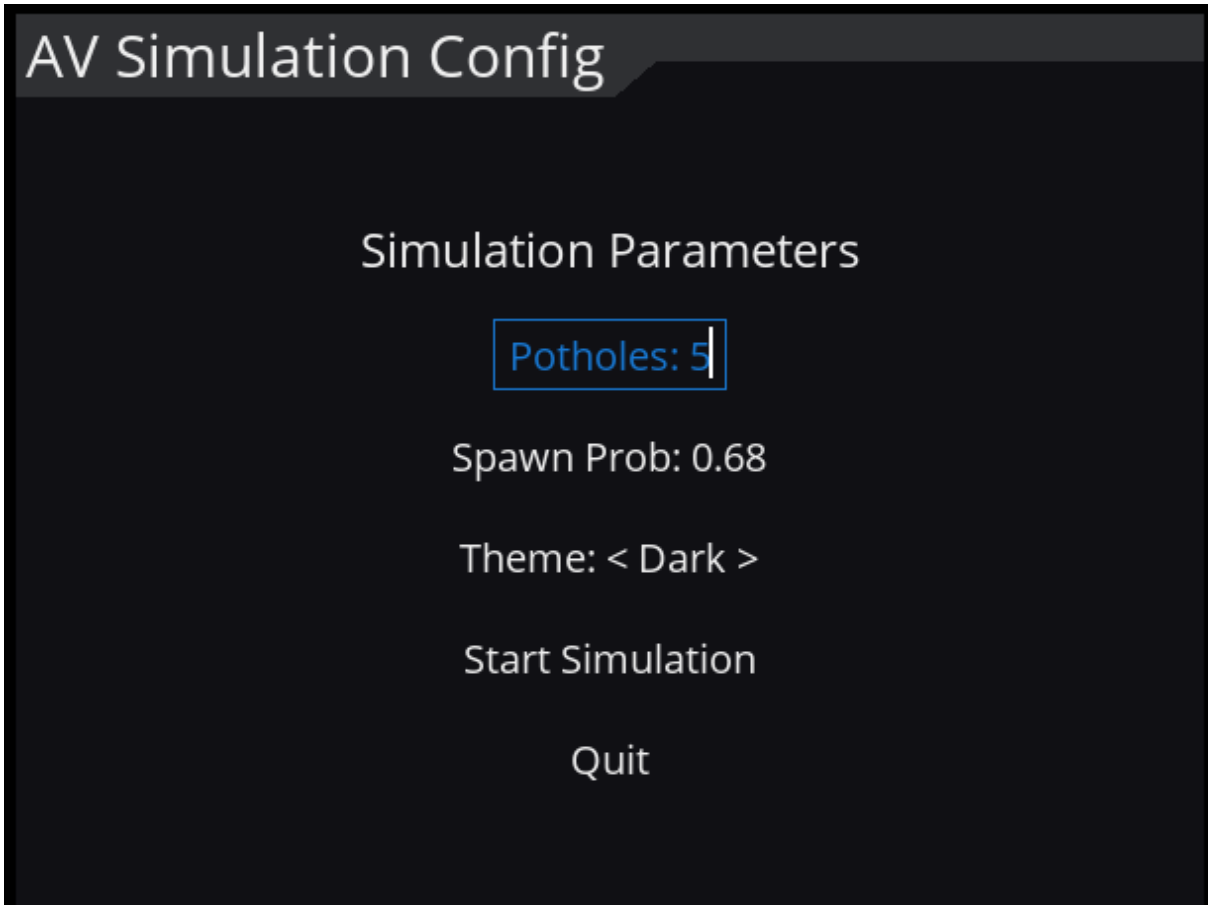
The collision detection system functions in a continuous cycle throughout the simulation tick, applying safety protocols through two separate phases: Predictive (prior to movement) and Reactive (following movement).

Algorithm 1: Predictive Safety Check (Pre-Movement)

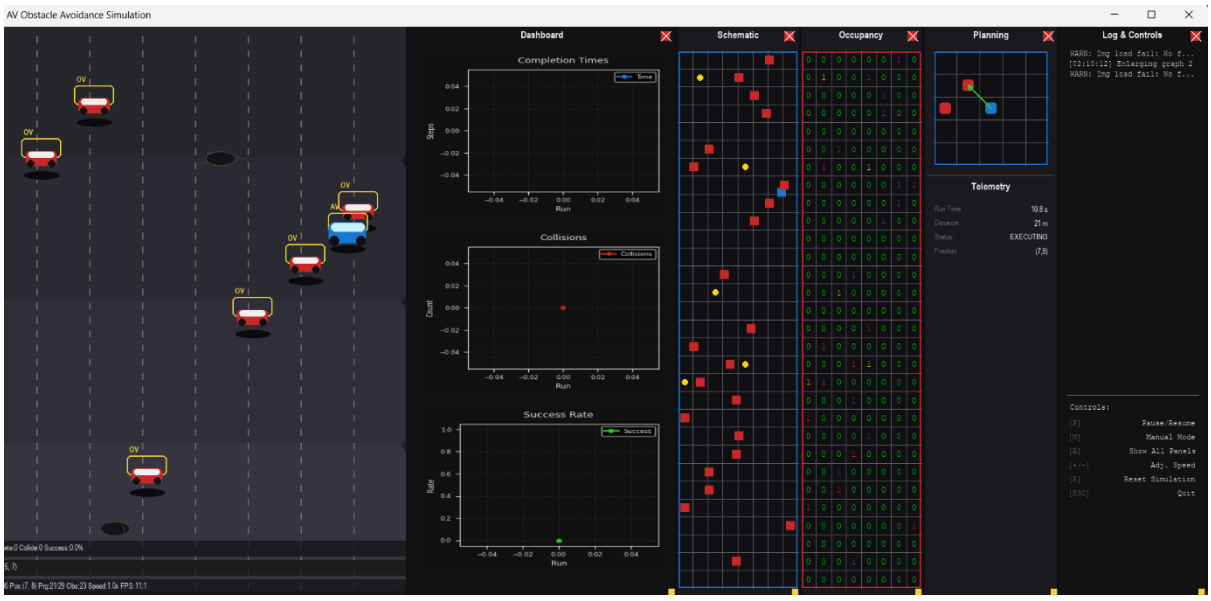
Input: Candidate Cell $C_{target}(x, y)$, Set of Potholes P, Set of Obstacles O.

Output: Boolean $IsSafe$.

1. **Boundary Validation:**
    - **If** $x < 0$ OR $x \geq GridWidth$ OR $y < 0$ OR $y \geq GridHeight$:
        - **Return** $False$ (Out of Bounds).
2. **Static Hazard Check:**
    - **If** $C_{target} \in P$ (Pothole Set):
        - **Return** $False$ (Pothole Detected).
3. **Dynamic Hazard Prediction:**
    - **For each** obstacle $O_i \in O$:
        - Get current position $Pos(O_i)$ and intended target $Target(O_i)$.
        - **If** $Target(O_i) == C_{target}$:
            - **Return** $False$ (Collision imminent: Space will be occupied).
        - **Else If** $Pos(O_i) == C_{target}$ **AND** $Target(O_i) == NULL$ (Stationary):
            - **Return** $False$ (Collision imminent: Hitting stationary vehicle).
4. **Clearance:**
    - **Return** $True$ (Path is Clear).



- Collision Detection Main menu.



- Collision detection Simulation Interface.

7Future Development

Transitioning from Deterministic to Probabilistic Perception

At present, the AV functions with "God-view" awareness, obtaining flawless information regarding obstacle locations and traffic congestion. Upcoming advancements will incorporate Sensor Uncertainty.

LIDAR Simulation: Rather than directly accessing the occupancy grid, the AV could emulate ray-casting to identify obstacles, resulting in "blind spots" and occlusion.

Sensor Fusion: Employing a Kalman Filter to determine the location of moving obstacles in situations where sensor data is unreliable or sporadic.

Integration of Reinforcement Learning (RL)

Although the existing Modified Dijkstra and Occupancy Checks are efficient rule-based systems, they lack the ability to "learn" from errors.

- RL Agent: The modular design enables substituting the av_decide_and_move function with a Deep Q-Network (DQN). The simulation serves as a gym setting where the agent earns rewards for speed and incurs penalties for collisions, ultimately mastering intricate behaviors such as overtaking or defensive driving without direct programming.

Intricate Urban Traffic Dynamics

The existing global planner considers nodes as basic junctions. Subsequent versions will include intricate traffic regulations:

- Traffic Signals: Utilizing state machines for traffic lights at key graph junctions
- V2V Communication: emulating Vehicle-to-Vehicle information sharing, enabling the AV to change course not only according to traffic density but also considering accidents reported by other simulated vehicles in front.

8.4 Shift to 3D Physic

The reasoning established in this context is independent of dimensions. An important future objective is to transfer the Map Engine and Path Planner into a 3D engine (e.g., Unity or Unreal Engine). This would enable the simulation of physical forces—friction, momentum, and banking—introducing a dimension of mechanical authenticity to the existing algorithmic reasoning.

8Conclusion

The project successfully demonstrates a comprehensive and multi-layered approach to autonomous vehicle (AV) navigation using Python, Pygame, OSMnx, and advanced path-planning algorithms. Across the two simulation environments—grid-based obstacle avoidance and real-world road network navigation—the system integrates perception, prediction, collision detection, and optimal route planning into a unified framework.

In the **grid-based simulation**, the AV is able to navigate through a dynamically changing environment containing moving obstacle vehicles and static potholes. Through predictive collision detection, a well-structured occupancy grid, and prioritized movement selection, the AV consistently avoids hazards and makes safe decisions. The use of animation, KPIs, and multiple visualization panels (schematic map, occupancy matrix, and planning map) provides a clear and intuitive understanding of the underlying algorithmic behaviour.

Building upon this foundation, the **OSM-based pathfinding system** extends the simulation to real-world urban road networks. By abstracting complex city road graphs, modelling traffic congestion using dynamic edge costs, and allowing waypoint-based navigation, the system mimics modern navigation behaviour used in real autonomous systems. The integration of real-time recalculations, manual road-blocking, and traffic-aware path planning significantly enhances the realism and applicability of the simulation. Together, these features demonstrate the scalability of the algorithm from a simple 8×30 grid to large, real-city networks.

Overall, the project achieves its core objectives: designing safe navigation policies, implementing reliable collision detection, enabling dynamic route planning, and producing high-quality multi-panel visualizations for analysis. Both simulations not only showcase the theoretical foundations of autonomous navigation but also provide an interactive platform for experimentation. This project can serve as a strong base for future enhancements, including sensor fusion, machine learning-based decision-making, and reinforcement learning-driven path optimization.

9.Reference

Academic Papers:

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*.

Fox, D., Burgard, W., & Thrun, S. (1997). "The Dynamic Window Approach to Collision Avoidance." *IEEE Robotics & Automation Magazine*.

LaValle, S. M. (2006). Planning Algorithms. Cambridge University Press.

Technical Documentation:

Pygame Documentation. Retrieved from https://www.pygame.org/docs/

Python Dataclasses. Retrieved from https://docs.python.org/3/library/dataclasses.html

Related Projects:

CARLA Simulator - Open-source simulator for autonomous driving research.

Apollo Auto - Baidu's open autonomous driving platform

OpenAI Gym - Toolkit for developing RL algorithms