Autonomous Vehicle Sensor Fusion and Alert System

This project simulates an autonomous vehicle's sensor system, performs real-time sensor fusion and path planning in C++, and visualizes detections with a PyQT GUI and PyTorch model in Python. It is designed as an educational-grade, two-week project.

Software architecture

- Concurrent Sensor Simulation: C++ threads for LiDAR, Camera, GPS; per□sensor rates; lock□free queues.
- Data Fusion & Processing: central fusion loop; detection matching; parallel obstacle checks; grid map.
- Path Planning & Alerts: A* replans; collision/off□course warnings; event logging.
- gRPC Link: C++ streams state, path, obstacles to Python.
- Python GUI & ML: PyQt live map; PyTorch classifier; dashboard metrics.

Folder Structure

```
project_root/
│
├── cpp/
│   ├── CMakeLists.txt
│   ├── main.cpp
│   ├── sensors/
│   │   ├── Sensor.h / .cpp
│   │   ├── LidarSensor.h / .cpp
│   │   ├── CameraSensor.h / .cpp
│   │   └── GPSSensor.h / .cpp
│   ├── fusion/
│   │   ├── SensorFusion.h / .cpp
│   ├── planning/
│   │   ├── PathPlanner.h / .cpp
│   ├── grpc_service/
│   │   ├── sensor_data.proto
│   │   ├── grpc_service.h / .cpp
│   ├── utils/
│   │   ├── BloomFilter.h / .cpp
│   │   ├── CountMinSketch.h / .cpp
│
├── python/
│   ├── requirements.txt
│   ├── app.py
│   ├── ui/
│   │   ├── main_window.py
│   │   ├── map_widget.py
│   │   └── sensor_panel.py
│   ├── grpc_client/
│   │   ├── sensor_client.py
│   │   ├── sensor_data_pb2.py
│   │   └── sensor_data_pb2_grpc.py
│   ├── model/
│   │   ├── detector.py
│   │   └── trainer.py
```

```
│   ├── config/
│   │   └── settings.py
│   └── utils/
│       └── data_processing.py
│
└── README.md
```

🧠 Module Descriptions and Responsibilities

C++ Modules

- cpp/main.cpp
  Initializes all sensors, starts sensor fusion, planning, and gRPC server.
- cpp/sensors/
  Houses all sensor simulation code (Lidar, Camera, GPS). Allows user-defined placement (front, back, etc.).
- cpp/fusion/
  Contains the `SensorFusion` module. Gathers data from all sensors, applies Bloom Filter and Count-Min Sketch for de-duplication and frequency tracking.
- cpp/planning/
  Implements A* search for path planning, uses a priority queue for route optimization, and generates alerts.
- cpp/grpc_service/
  Handles gRPC communication. Streams fused sensor data to the Python side. Uses `sensor_data.proto` for message schemas.
- cpp/utils/
  Implements the core data structures:
  - BloomFilter: For detecting duplicate sensor events.
  - CountMinSketch: For estimating frequency of object detection or obstacle appearance.

Python Modules

- python/app.py
  Entry point for launching the PyQT UI and model integration.
- python/ui/
  - main_window.py: Core GUI window.
  - map_widget.py: Displays vehicle position and detections on a 2D map.
  - sensor_panel.py: Interface to configure sensors and their positions.
- python/grpc_client/
  - sensor_client.py: Connects to C++ gRPC service and receives real-time sensor fusion data.
  - sensor_data_pb2.py / sensor_data_pb2_grpc.py: Auto-generated gRPC bindings.
- python/model/

  - detector.py: Defines a PyTorch-based detection/classification model.
  - trainer.py: (Optional) Adds training and fine-tuning capabilities.
- python/config/
  - settings.py: Configurable values like sensor types, vehicle dimensions, etc.
- python/utils/
  - data_processing.py: Converts gRPC data into formats usable by PyTorch and UI modules.

Data Structure Usage

Data Structure	Location	Purpose
Bloom Filter	`cpp/utils/BloomFilter.cpp` → `SensorFusion.cpp`	Prevents processing the same obstacle multiple times.
Count-Min Sketch	`cpp/utils/CountMinSketch.cpp` → `SensorFusion.cpp`	Tracks frequency of object detections or hazard appearances.
Priority Queue	`cpp/planning/PathPlanner.cpp`	Manages open set in A* pathfinding for vehicle routing.
LRU Cache (Optional)	`cpp/utils/Cache.cpp`	Stores recent fusion results or paths for quick re-use.

14-Day Development Plan

Week 1: C++ Core System

- Day 1–2: Setup project structure, CMake, Python venv, sensor configuration planning.
- Day 3: Build abstract sensor base class.
- Day 4: Implement Lidar, Camera, GPS sensor classes.
- Day 5: Build `SensorFusion` module and data combination logic.
- Day 6: Implement and test `BloomFilter` and `CountMinSketch`.
- Day 7: Build path planning (A*) and alerting system using priority queue.

Week 2: Integration and Python System

- Day 8: Design and implement C++ gRPC server and `.proto` file.
- Day 9: Set up Python UI base with PyQt5: main window and map widget.
- Day 10: Develop Python gRPC client, receive data from C++ server.
- Day 11: Implement PyTorch model to analyze sensor fusion outputs.
- Day 12: Add sensor configuration UI panel and physical properties editor.
- Day 13: Full integration test of the entire pipeline: simulation → fusion → gRPC → model → UI.
- Day 14: Final polish: error handling, logging, code cleaning, documentation, test cases.