كلية العلوم و التقنيات بطنجة
Faculté des Sciences et Techniques de Tanger

جامعة عبد المالك السعدي
Université Abdelmalek Essaâdi

LST : IDAI

1/2 Module : POO en C++

RAPPORT DE PROJET :

SIMULATEUR D'UNE SMART CITY EN C++ (POO & Raylib)

SOUS-PROJET 1 :

TRAFFIC CORE

Réalisé par :

- EL BOURMAKI Salim
- EL HAJIOUI Houssam
- SADIKI Maroua
- DANY Homam

Sous l'encadrement de :

Pr. BEN ABDEL OUAHAB Ikram

Année universitaire : 2025-2026

SOMMAIRE :

Abstract

In the context of rapidly growing urban environments, efficient traffic management has become a paramount challenge for modern cities. This project introduces "Traffic Core," an advanced real-time traffic monitoring and analysis system designed to serve as the backbone of an Intelligent Transportation System.

Leveraging the power of Computer Vision and Deep Learning, the proposed solution utilizes YOLOv8 for robust object detection and ByteTrack for accurate multi-object tracking. The system is capable of performing critical tasks such as vehicle counting, classification, and speed estimation with high precision across varying traffic conditions. The processed data is visualized through an interactive dashboard, providing operators with actionable insights and Key Performance Indicators (KPIs) for decision-making.

Beyond standalone monitoring, this project establishes the foundational "Ground Truth" required for a holistic smart city ecosystem. It is designed to integrate seamlessly with parallel sub-projects, including Adaptive Signal Control and Dynamic Routing, thereby enabling a fully synchronized and responsive urban traffic network.

Keywords: **Computer Vision, YOLOv8, ByteTrack, Object Detection, Vehicle Counting, Speed Estimation, Traffic Analytics.**

1. Introduction :

Dans un contexte d'urbanisation croissante et de modernisation des infrastructures, le concept de "Smart City" (Ville Intelligente) s'impose comme une réponse incontournable pour optimiser la gestion des flux urbains. La simulation informatique joue un rôle clé dans cette transformation, permettant de modéliser, visualiser et comprendre les dynamiques complexes de la circulation routière.

C'est dans cette optique que s'inscrit le projet "Traffic Core". Il s'agit d'un moteur de simulation de trafic routier développé en C++, conçu pour gérer de manière autonome une flotte hétérogène de véhicules. L'objectif principal n'est pas seulement d'afficher des voitures, mais de simuler des comportements distincts et réalistes pour différents types d'entités : des Motos qui s'inclinent dans les virages, des Taxis avec leur signalisation propre, des Véhicules de Police en intervention et des Bus encombrants.

D'un point de vue technique, ce projet est une mise en application avancée de la Programmation Orientée Objet (POO). Le développement, réalisé sous l'environnement **VSCode** avec le système de construction **CMake**, s'appuie sur la bibliothèque **Raylib** pour assurer un rendu **3D** fluide et performant. Au cœur du système, une approche algorithmique basée sur la théorie des graphes permet de modéliser le réseau routier, offrant aux véhicules une capacité de navigation intelligente d'un nœud à un autre.

Ce rapport détaille l'architecture logicielle modulaire du projet, justifie les choix de conception (notamment l'usage du polymorphisme pour gérer la diversité des véhicules) et présente les scénarios de simulation mis en œuvre.

1.2 Environnement de Développement :

La réalisation de ce projet s'est appuyée sur une chaîne d'outils (toolchain) moderne et performante, choisie pour garantir la productivité du développement et la robustesse du code :

Visual Studio Code (VSCode) : Utilisé comme Environnement de Développement Intégré (IDE). Ce choix se justifie par sa légèreté et son extensibilité, offrant des outils puissants pour l'édition de code C++, l'auto-complétion (IntelliSense) et le débogage intégré.

Raylib : Bibliothèque graphique choisie pour le moteur de rendu. Contrairement aux moteurs de jeu lourds (Unity/Unreal), Raylib offre une approche "bas niveau" en C++ pur. Elle permet une gestion fine de la fenêtre, des entrées utilisateurs et du rendu 3D via OpenGL, ce qui est idéal pour comprendre les mécanismes internes d'une simulation.

CMake : Système de construction (Build System) multiplateforme. Il a permis d'automatiser le processus de compilation, de gérer les dépendances (notamment l'édition de liens avec Raylib) et de configurer le projet de manière standardisée, assurant sa portabilité.

GitHub : Plateforme de gestion de versions (VCS) basée sur Git. Utilisée pour le suivi des modifications du code source, la gestion des sauvegardes et l'historique du développement, garantissant ainsi la sécurité et la traçabilité du projet.

1.3 Objectifs du Module "TRAFFIC CORE" :

Ce projet s'inscrit dans le développement d'un simulateur de Smart City en C++. Notre groupe a été chargé du module "TRAFFIC CORE". Considéré comme la colonne vertébrale du simulateur, ce module a pour responsabilités critiques :

- **La Modélisation Topologique :** Représenter un réseau routier réaliste (routes droites, courbes, intersections) sous une forme mathématique exploitable.
- **La Gestion des Agents :** Simuler le déplacement autonome de véhicules variés (voitures, bus, camions, police… etc.) respectant les règles de circulation.
- **La Visualisation Temps Réel :** Offrir un rendu 3D fluide via la bibliothèque **Raylib.**
- **L'Architecture Modulaire :** Fournir une base de code flexible et configurable (vitesse, densité) pour l'intégration future d'autres modules.

1.4 Problématique et Approche Technique :

Le défi principal réside dans la gestion de la complexité entre la logique mathématique et le rendu visuel : **Comment faire naviguer des véhicules de manière fluide sur des routes courbes tout en utilisant une structure de données discrète ?**

- Pour y répondre, nous avons adopté une approche rigoureuse basée sur la Programmation Orientée Objet (POO) en C++, structurée autour :
- D'un Graphe Orienté pour la logique de navigation.
- De l'Héritage et du Polymorphisme pour traiter uniformément les types de véhicules.
- De Design Patterns (État, Singleton) pour séparer la logique de simulation du rendu graphique.

2. Architecture et Conception Technique :

L'architecture logicielle du projet **"Traffic Core"** a été pensée pour répondre aux exigences de performance d'une simulation temps réel tout en respectant les principes de la conception modulaire. L'objectif était de créer un système extensible où l'ajout de nouveaux types de véhicules ou de règles de circulation n'impacte pas la stabilité du noyau existant.

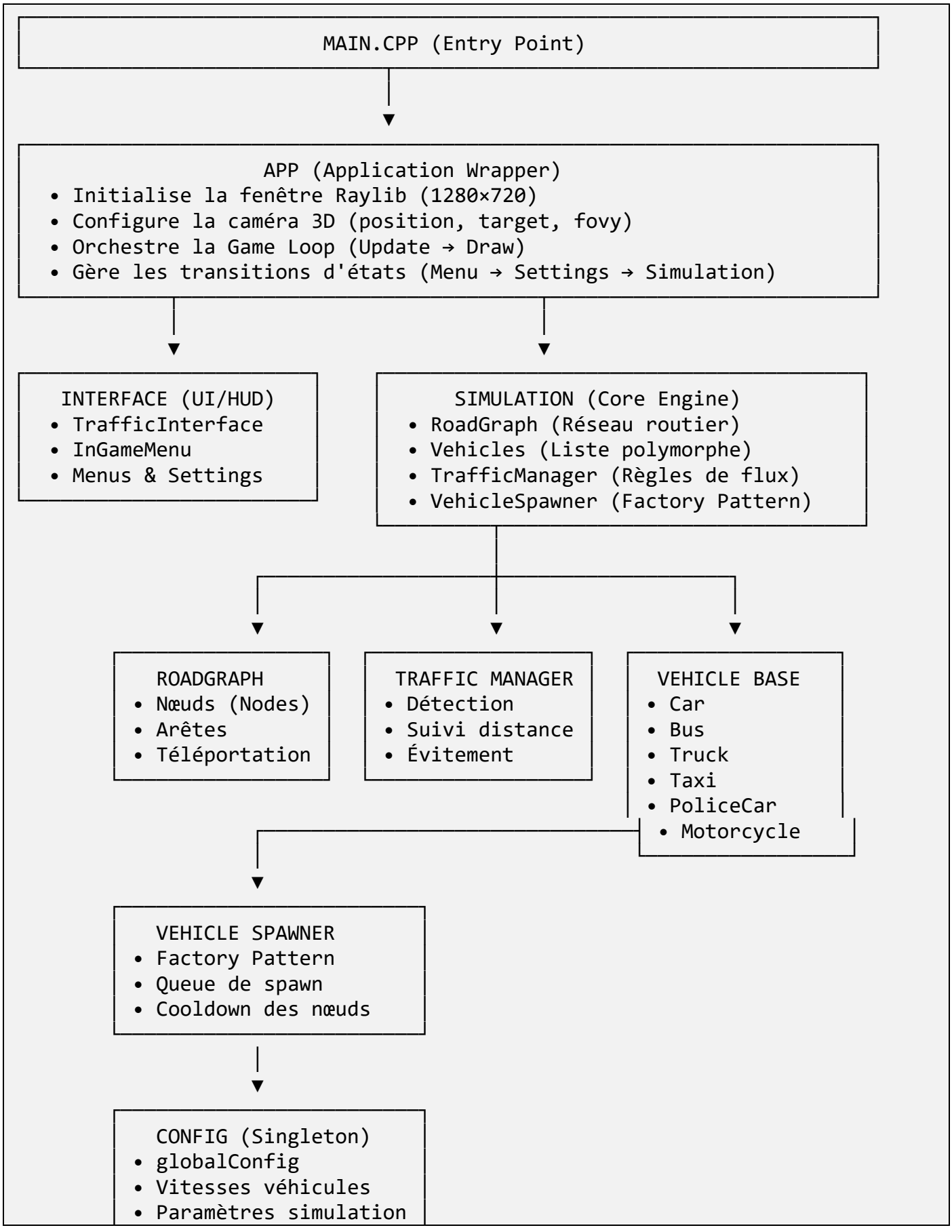
2.1 Organisation des fichiers (Code Source) :

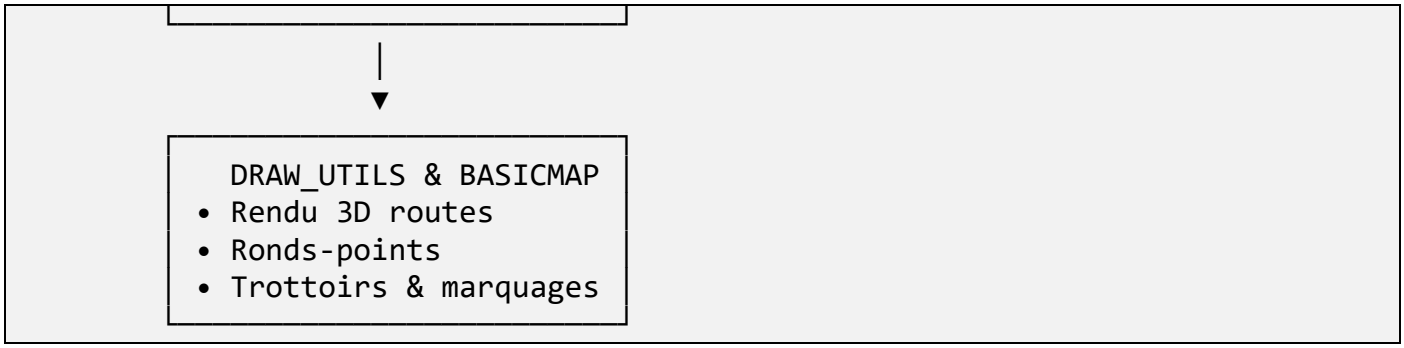
Le projet suit une structure standardisée séparant clairement les déclarations (headers), les implémentations (sources) et les scripts de construction. Cette séparation favorise la lisibilité et le travail de maintenance.

```
TrafficCore/
├── include/                    # Header files (Declarations)
│   ├── app.h                   # Main application wrapper & lifecycle
│   ├── basicmap.h              # Static map geometry and rendering
│   ├── config.h                # Global configuration & constants
│   ├── draw_utils.h            # Helper functions for 3D drawing
│   ├── ingame_menu.h           # Logic for the pause/settings menu
│   ├── interface_new.h         # Main UI & HUD system
│   ├── roadgraph.h             # Graph logic (nodes/edges) for pathfinding
│   ├── simulation.h            # Core simulation loop & state
│   ├── spawner.h               # Vehicle generation logic
│   ├── traffic_manager.h       # Global traffic flow control
│   └── vehicle.h               # Vehicle physics, AI & collision logic
│
├── src/                        # Source files (Implementations)
│   ├── main.cpp                # Entry point
│   ├── app.cpp                 # Application setup and loop handling
│   ├── basicmap.cpp            # Map visualization logic
│   ├── config.cpp              # Config loader
│   ├── draw_utils.cpp          # Rendering helpers
│   ├── ingame_menu.cpp         # Menu interactivity implementation
│   ├── interface_new.cpp       # UI rendering implementation
│   ├── roadgraph.cpp           # Road network calculations
│   ├── simulation.cpp          # Main simulation engine logic
│   ├── spawner.cpp             # Vehicle spawning algorithms
│   ├── traffic_manager.cpp     # Traffic rules & coordination
│   └── vehicle.cpp             # Vehicle behavior implementation
│
├── screenshots/                # Project visual showcase
│   ├── hero_shot.png
│   ├── ui_settings.png
│   └── vehicles_closeup.png
│
├── demos/                      # Video Demonstrations
│   └── demo.mp4                # Simulation features showcase
│
├── tests/                      # Testing suite
│   └── unit_tests.cpp          # Unit tests for core components
│
├── Rapport TRAFFIC_CORE.pdf    # Final Project Report
├── Makefile                    # Build instructions
└── README.md                   # Project documentation
```

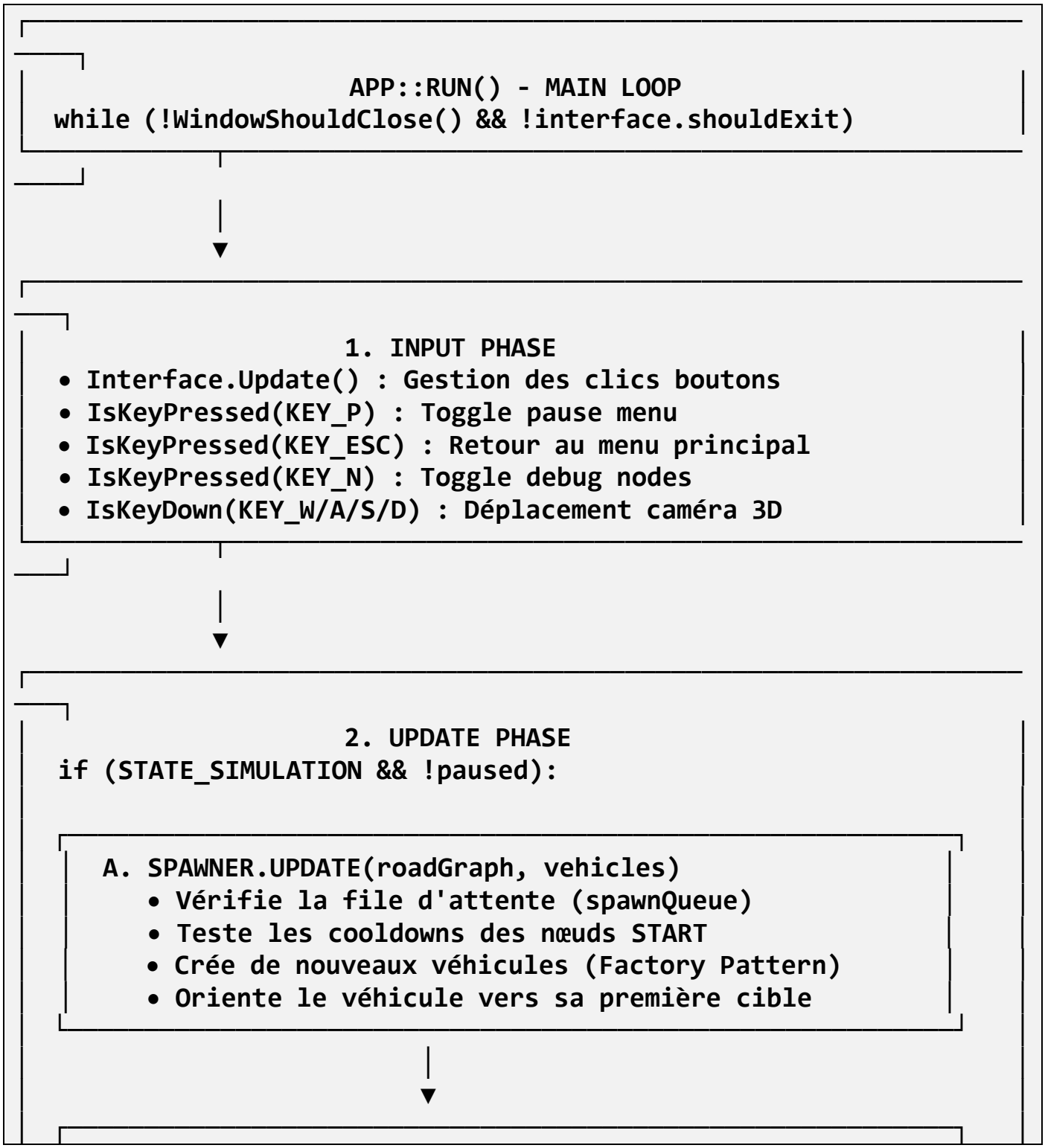
2.2 Diagramme d'Architecture Globale

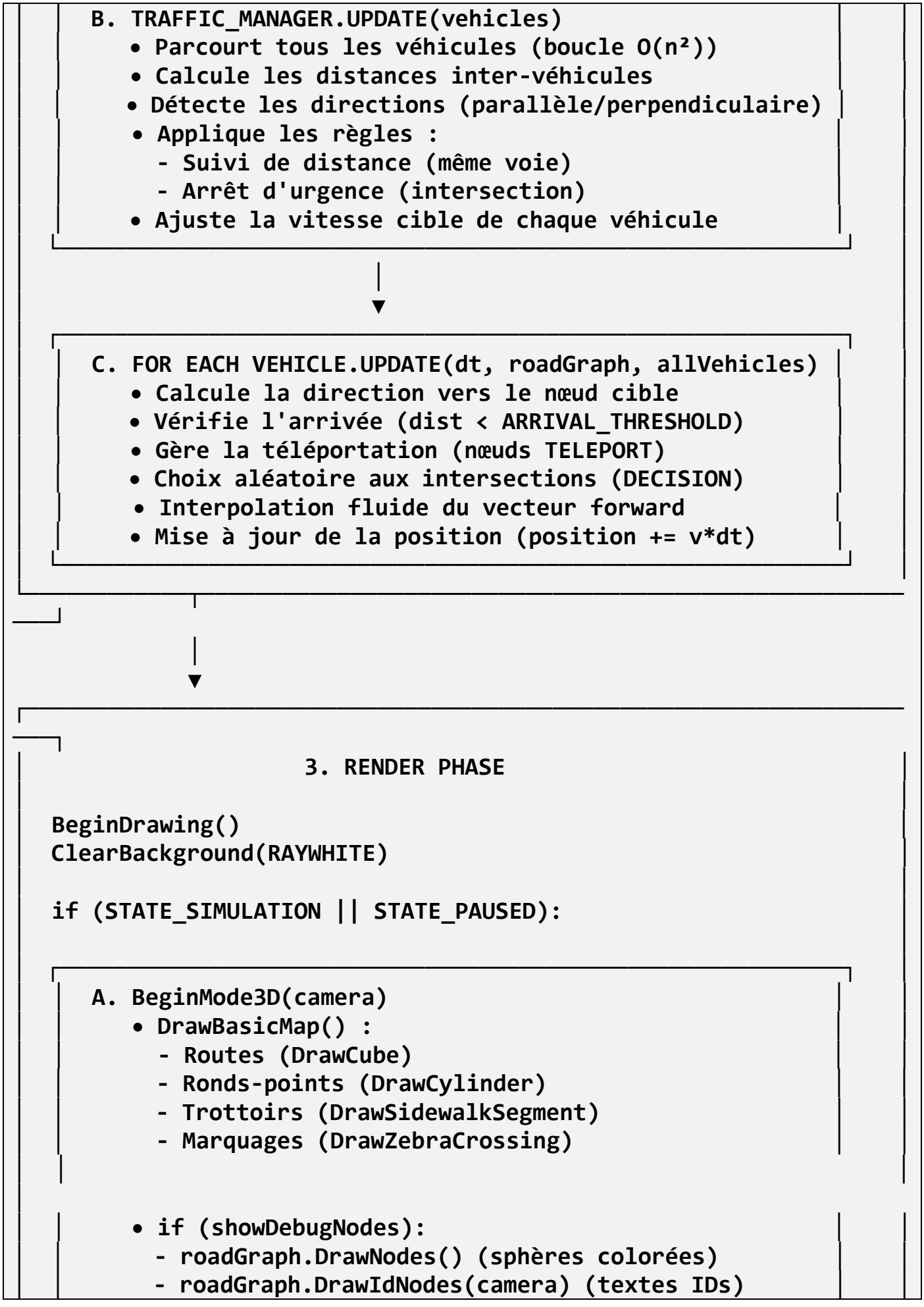
Le système est organisé en couches modulaires avec séparation claire des responsabilités :





2.3 Flux de données :




B. TRAFFIC_MANAGER.UPDATE(vehicles)
• Parcourt tous les véhicules (boucle O(n²))
• Calcule les distances inter-véhicules
• Détecte les directions (parallèle/perpendiculaire)
• Applique les règles :
- Suivi de distance (même voie)
- Arrêt d'urgence (intersection)
• Ajuste la vitesse cible de chaque véhicule
C. FOR EACH VEHICLE.UPDATE(dt, roadGraph, allVehicles)
• Calcule la direction vers le nœud cible
• Vérifie l'arrivée (dist < ARRIVAL_THRESHOLD)
• Gère la téléportation (nœuds TELEPORT)
• Choix aléatoire aux intersections (DECISION)
• Interpolation fluide du vecteur forward
• Mise à jour de la position (position += v*dt)
3. RENDER PHASE
BeginDrawing()
ClearBackground(RAYWHITE)
if (STATE_SIMULATION || STATE_PAUSED):
A. BeginMode3D(camera)
• DrawBasicMap() :
- Routes (DrawCube)
- Ronds-points (DrawCylinder)
- Trottoirs (DrawSidewalkSegment)
- Marquages (DrawZebraCrossing)
• if (showDebugNodes):
- roadGraph.DrawNodes() (sphères colorées)
- roadGraph.DrawIdNodes(camera) (textes IDs)

- FOR EACH vehicle->draw() :
  - Car : Châssis + Cabine + 4 Roues + Phares
  - Bus : Corps allongé + Fenêtres + 4 Roues
  - Truck : Cabine + Remorque + 6 Roues
  - Taxi : Voiture + Panneau clignotant
  - PoliceCar : Voiture + Gyrophare alterné
  - Motorcycle : Châssis + Inclinaison + 2 Roues

EndMode3D()

▼

B. 2D OVERLAYS (HUD)
- DrawText("Traffic Core Simulator")
- DrawText("P : Settings")
- DrawText("ESC : Main Menu")
- DrawText("N : Show Nodes")
- DrawText("Vehicles: X")

- if (pauseMenu.isVisible):
  - InGameMenu.Draw() (panneau de configuration)

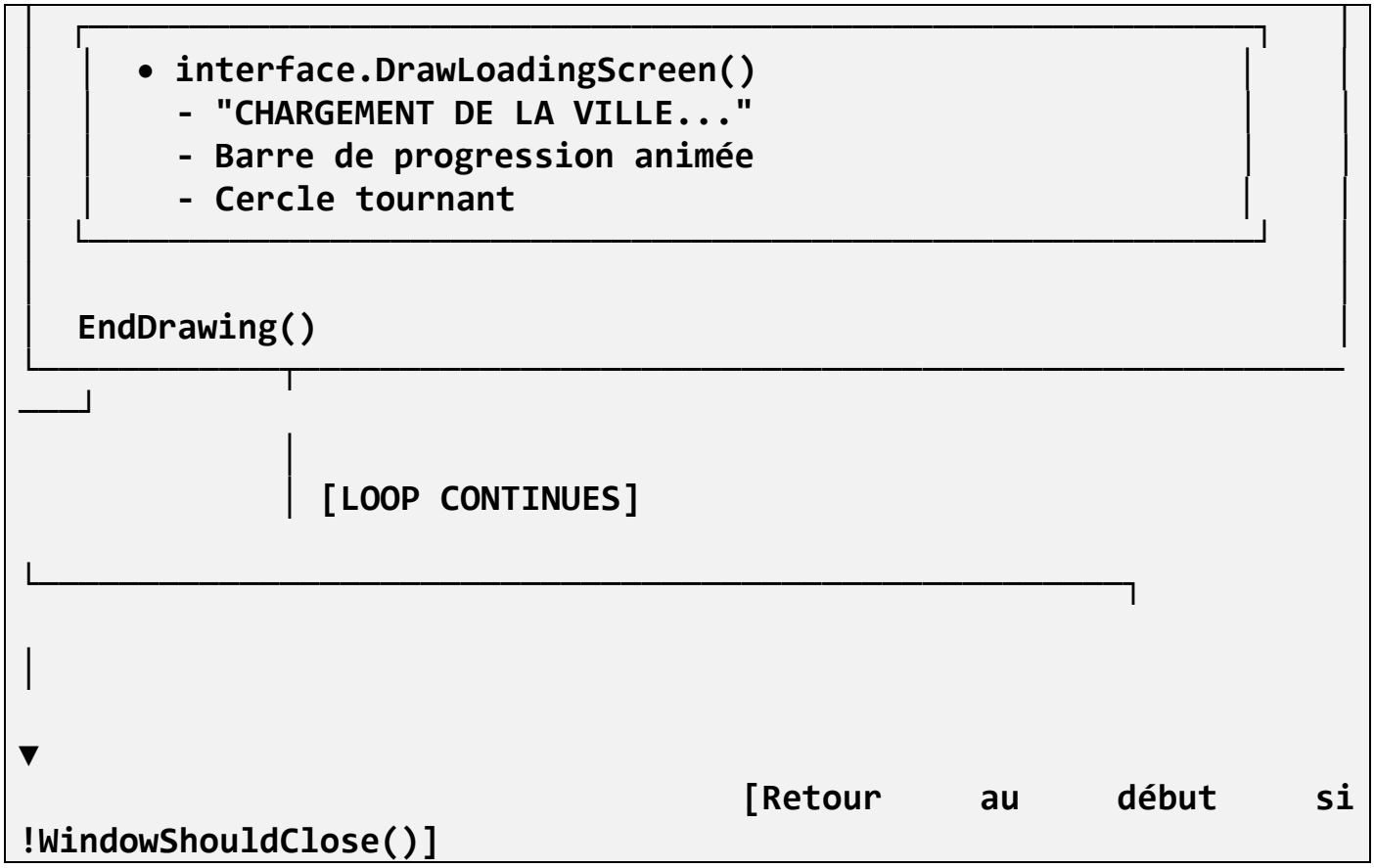
else if (STATE_MENU):

- interface.DrawMainMenu()
  - Fond animé avec grille
  - Logo "TRAFFIC CITY"
  - Boutons (DÉMARRER, QUITTER)

else if (STATE_SETTINGS):

- interface.DrawSettingsMenu()
  - Max Vehicles (boutons +/-)
  - Speed Slider (0.5x à 3.0x)
  - Liste véhicules (Car, Bus, Truck...)
  - Boutons (RETOUR, LANCER SIMULATION)

else if (interface.shouldStartSimulation && !gameStarted):



2.4 Description des Modules Principaux :

L'application repose sur le patron de conception **Composite**, où l'objet principal App agrège les sous-systèmes essentiels :

- **Le Moteur (App) :** Il agit comme le conteneur principal. Il initialise la fenêtre Raylib, configure la caméra 3D et gère la boucle principale (*Input -> Update -> Draw*). Il ne contient aucune logique de trafic propre, déléguant cette tâche à la Simulation.
- **Le Noyau de Simulation (Simulation) :** C'est le "cerveau" du projet. Il possède deux composants majeurs :
  - ✓ Le **Réseau Routier (RoadGraph)** : Une représentation mathématique de la ville sous forme de graphe (Nœuds et Arêtes), permettant le calcul d'itinéraires.
  - ✓ La **Flotte de Véhicules** : Un conteneur dynamique gérant toutes les entités en mouvement.
- **Les Entités (Vehicle et dérivés) :** Chaque agent de la simulation est une instance d'une classe dérivée de Vehicle. Chaque véhicule est autonome : il connaît sa position, sa vitesse et sa destination immédiate sur le graphe.

2.5 Justification des Choix POO et Design Patterns :

La robustesse du projet repose sur l'utilisation avancée des concepts de la Programmation Orientée Objet (POO). Voici l'analyse des choix techniques effectués dans le code :

a. Héritage et Polymorphisme (Le point fort du projet) ;

Au lieu de créer une classe unique "Véhicule" avec des conditions complexes (ex : if type == MOTO), nous avons utilisé l'héritage pour spécialiser les comportements.

- **Classe Abstraite Vehicle :** Elle définit le contrat commun (méthodes update et draw virtuelles).
- **Spécialisation par Surcharge (override) :**
  - ✓ **La Moto (Motorcycle) :** Surcharge la méthode draw() pour inclure une physique d'inclinaison (tiltAngle) dans les virages, calculée dynamiquement selon le vecteur de direction.
  - ✓ **Le Taxi (Taxi) :** Surcharge update() pour gérer un minuteur indépendant (signBlinkTimer) faisant clignoter son enseigne lumineuse.
  - ✓ **La Police (PoliceCar) :** Intègre une logique de sirène propre.

Ce choix permet d'ajouter un nouveau type de véhicule (ex : Ambulance) sans modifier une seule ligne du moteur de simulation existant (principe **Open/Closed** du SOLID).

b. Gestion de la Mémoire et Pointeurs Intelligents (RAII) ;

Le C++ moderne a été privilégié pour éviter les fuites de mémoire. Dans simulation.h, nous utilisons : std::vector<std::unique_ptr<Vehicle>> vehicles;

L'utilisation de std::unique_ptr garantit que la mémoire allouée à un véhicule est automatiquement libérée dès que celui-ci est détruit ou retiré de la simulation, éliminant les risques de *Memory Leaks* classiques en C++.

c. Encapsulation des Données ;

Les attributs sensibles sont protégés (private ou protected). Par exemple, l'angle d'inclinaison d'une moto ou l'état du minuteur d'un taxi ne sont pas accessibles depuis l'extérieur. Seules les méthodes publiques permettent d'interagir avec l'objet, garantissant l'intégrité de l'état du véhicule à chaque instant de la simulation.

3. Fonctionnalités et Scénario de Simulation :

Cette partie décrit les capacités fonctionnelles du moteur **Traffic Core** et présente un scénario d'exécution typique démontrant la robustesse de l'architecture POO.

3.1 Fonctionnalités Principales :

Le simulateur offre un ensemble de fonctionnalités techniques permettant la gestion d'un trafic urbain virtuel :

a. Navigation Autonome par Graphe :

Contrairement à une animation scriptée, les véhicules ne suivent pas une ligne prédéfinie.

- **Fonctionnement :** Le système utilise la classe RoadGraph pour définir des nœuds (intersections) et des arêtes (routes).
- **Intelligence :** Chaque véhicule calcule sa direction vers le nœud cible (targetNode) et ajuste sa rotation en temps réel pour suivre la route parfaitement. Lorsqu'un véhicule atteint un nœud, il interroge le graphe pour connaître les connexions sortantes et choisit sa prochaine destination aléatoirement, garantissant une circulation infinie et non répétitive.

b. Comportements Véhiculaires Spécifiques (Polymorphisme Visuel) :

Grâce à l'architecture héritée de Vehicle, chaque type d'entité possède une signature visuelle et physique unique :

- **Physique de la Moto :** La classe Motorcycle implémente un calcul d'inclinaison (tiltAngle). Plus le virage est serré, plus la moto s'incline, simulant la force centrifuge.
- **Animation des Taxis :** La classe Taxi gère un état interne (timer) pour faire clignoter son enseigne lumineuse ("Taxi Sign") indépendamment du reste du trafic.
- **Véhicules de Service :** La classe PoliceCar intègre des éléments visuels distinctifs (gyrophares).

c. Caméra et Rendu 3D Interactif :

L'utilisateur n'est pas passif. La classe App intègre une Camera3D interactive :

- **Navigation :** Possibilité de se déplacer librement dans la scène (Zoom, Pan, Rotate) pour observer le trafic sous tous les angles.
- **Interface HUD :** Affichage en temps réel des statistiques via TrafficInterface (nombre de véhicules, FPS, temps de simulation).

d. Gestion du Cycle de Vie (Spawning) :

Le système gère dynamiquement l'apparition et la disparition des véhicules pour maintenir une densité de trafic constante sans saturer la mémoire, grâce au Spawner et à la gestion des pointeurs intelligents.

3.2 Exemple de Scénario Simulé :

Pour valider le bon fonctionnement du moteur, nous avons défini un scénario de test standard intitulé **"Flux Urbain Mixte"**.

⇒ **Objectif :** Vérifier que différents types de véhicules peuvent coexister sur le même graphe routier tout en exécutant leurs comportements spécifiques.

Étape 1 : Initialisation

- **Action :** Lancement de l'application via `App::Run()`.
- **Résultat :** La fenêtre s'ouvre, le réseau routier (`RoadGraph`) est généré. La caméra se place en vue plongeante. Le compteur de véhicules est à 0.

Étape 2 : Injection du Trafic

- **Action :** Le `Spawner` génère une première vague de véhicules.
- **Observation :**
  - ✓ Une **Voiture de Police** apparaît au Nœud A et se dirige vers le Nœud B.
  - ✓ Un **Taxi** apparaît au Nœud C.
  - ✓ Une **Moto** apparaît au Nœud D.

Étape 3 : Comportement en Virage (Test de la Moto)

- **Situation :** La Moto arrive à une intersection et doit tourner à 90 degrés vers la droite.
- **Observation Visuelle :** Au moment de la rotation, le modèle 3D de la moto s'incline (`tiltAngle` devient négatif). Une fois la ligne droite reprise, la moto se redresse progressivement.
- **Validation Technique :** La méthode `Motorcycle::draw()` a correctement interprété le changement de vecteur directionnel.

Étape 4 : Animation d'État (Test du Taxi)

- **Situation :** Le Taxi circule en ligne droite.
- **Observation Visuelle :** L'enseigne jaune sur le toit s'allume et s'éteint régulièrement.
- **Validation Technique :** La méthode `Taxi::update()` a correctement incrémenté `signBlinkTimer` sans bloquer le mouvement du véhicule.

Étape 5 : Interruption et Menu

- **Action :** L'utilisateur appuie sur la touche de menu (**P** ou **Echap**).
- **Résultat :** La simulation se fige (le `dt` passe à 0). Le menu `InGameMenu` s'affiche en superposition, permettant de reprendre ou de quitter.

Conclusion du Scénario :

Ce scénario confirme que l'architecture polymorphe (polymorphisme) fonctionne : la boucle principale traite tous les véhicules comme des Vehicle* génériques, mais chacun réagit selon sa propre nature (*Moto*, *Taxi*, *Police, etc.*) à l'écran.

4. Interface Utilisateur (UI) :

4.1 Objectifs de l'Interface :

L'interface a été conçue selon trois axes principaux :

- **Accessibilité :** Permettre une prise en main intuitive sans formation préalable ;
- **Flexibilité :** Offrir un contrôle granulaire des paramètres de simulation ;
- **Immersion :** Proposer une expérience visuelle moderne et engageante.

L'architecture UI s'articule autour de trois phases distinctes : le Menu Principal, la Configuration Avancée, et l'Environnement de Simulation en temps réel.

NB1 : Pour lancer, il faut cliquer ctrl+f5 dans le vscode.

4.2 ARCHITECTURE DE L'INTERFACE :

4.3 Menu Principal (Écran d'Accueil) :

C'est le point d'entrée de l'application. Il a été conçu pour établir immédiatement l'identité visuelle du simulateur tout en validant le bon fonctionnement du moteur graphique.

⇒ **Éléments visuels :**

- **Titre "TRAFFIC CITY" :** Affiché avec une typographie stylisée, accompagné d'effets de particules flottantes et d'une grille dynamique en arrière-plan pour confirmer l'initialisation correcte du moteur de rendu 3D

- **Animation centrale :** Une voiture jaune stylisée en rotation constante servant d'aperçu thématique ;

○ ***NB2 : Il faut utiliser la souris ou pour naviguer !***

⇒ **Contrôles interactifs :**

- **Bouton [DÉMARRER] :** Transition vers l'écran de configuration avec effet de survol (hover) changeant la couleur pour indiquer la cliquabilité ;
- **Bouton [QUITTER] :** Fermeture propre de l'application avec libération correcte des ressources Raylib.

⇒ **Implémentation technique :** Ce menu est géré par la classe MainMenu qui hérite du système de states pattern, permettant une transition fluide vers le mode configuration sans réinitialiser le contexte graphique.

4.4 Menu de Configuration (Paramétrage Pré-Simulation) :

Cette interface constitue le "cerveau" du système. Elle permet de définir tous les paramètres de charge avant la génération procédurale de la ville et l'instanciation des véhicules.

a. Contrôle des Limites et Vitesse de Simulation :

Limite Maximale de Véhicules :

- **Fonction1 :** Définit le plafond total d'entités simultanées autorisées sur le réseau routier

- **Contrôles :** Boutons incrémentaux **[+]** et **[-]** permettant d'ajuster la valeur entre la somme actuelle des véhicules configurés et un maximum de 100 unités
- **Validation :** Un système de vérification empêche de descendre en dessous du total configuré pour éviter les incohérences

Curseur de Vitesse de Simulation :

- **Fonction2 :** Règle le facteur de dilatation temporelle (time scale) de la boucle de jeu
- **Plage :** De 0.5x (ralenti pour analyse fine des comportements) à 3.0x (accéléré pour observer la formation de congestions à long terme)

⇒ **Implémentation :** Le multiplicateur de vitesse est appliqué au `deltaTime` (`dt`) dans la méthode `Simulation::update(float dt)` ;

b. Gestion du circulation Auto-Mobile :

L'utilisateur dispose d'un contrôle granulaire sur la composition de la flotte circulante. Six types de véhicules sont disponibles, chacun correspondant à une classe dérivée de `Vehicle` dans le code C++ :

Types de Véhicules :

- **Voiture (Car) :** Véhicule standard, classe de base pour les comportements génériques ;
- **Bus :** Véhicule volumineux avec animation de portes ;
- **Camion (Truck)** : Entité lourde avec physique de remorque ;
- **Taxi :** Intègre un système de clignotement de l'enseigne lumineuse via `signBlinkTimer` ;
- **Police (PoliceCar) :** Véhicule de service avec gyrophares animés
- **Moto (Motorcycle) :** Implémente une physique d'inclinaison dynamique (`tiltAngle`) dans les virages ;

Interface de Contrôle :

- **Boutons unitaires [+] et [-] :** Ajustement individuel de chaque type avec désactivation automatique si la limite maximale est atteinte
- **Barre d'utilisation :** Indicateur visuel en temps réel affichant le ratio Total configuré / Max Limit
- **Vert :** Capacité disponible
- **Rouge :** Saturation atteinte (empêche tout ajout supplémentaire)

⇒ **Implémentation technique :** Cette interface communique directement avec la classe `VehicleSpawner` qui utilise un pattern `Factory` pour instancier les bons types polymorphes via `std::unique_ptr<Vehicle>`.

c. Navigation Inter-Menus :

- **Bouton [RETOUR] :** Annule les modifications en cours et revient au menu principal sans sauvegarder

- **Bouton [LANCER SIMULATION] :** Valide les paramètres, les transmet à Simulation, et déclenche l'écran de chargement avec génération du RoadGraph

4.5 ENVIRONNEMENT DE SIMULATION (MODE IN-GAME) :

Une fois la simulation lancée, l'utilisateur bascule dans un environnement 3D interactif où tous les véhicules configurés circulent de manière autonome sur le réseau routier généré.

a. HUD (Heads-Up Display) - Commandes Clavier:

Un overlay semi-transparent affiché en permanence en haut à gauche de l'écran rappelle les raccourcis disponibles :

Raccourcis Clavier :

- **[P] Paramètres :** Ouvre un menu de pause superposé (Overlay) permettant de modifier dynamiquement les paramètres sans quitter la simulation ni réinitialiser l'état des véhicules ;
- **[ESC] Menu Principal :** Retour immédiat à l'écran d'accueil avec arrêt de la simulation ;
- **[N] Show Nodes :** Active/désactive l'affichage des nœuds du graphe routier (sphères colorées aux intersections) pour le débogage des trajectoires ;
- **[WASD] Move Camera :** Contrôle de déplacement libre dans la scène 3D via la Camera3D de Raylib, permettant d'observer différents carrefours sous tous les angles ;

⇒ **Implémentation technique :** Ces contrôles sont gérés dans la boucle principale `App::handleInput()` qui dispatche les événements vers les sous-systèmes appropriés (*Simulation, Camera, DebugRenderer*).

b. Interactions Avancées à la Souris :

Click Car (Force Move) :

Fonction : Fonctionnalité de débogage permettant de cliquer sur un véhicule spécifique pour influencer son comportement ou débloquer une situation de congestion

⇒ **Implémentation :** Utilise un ray-casting 3D (`GetMouseRay`) pour détecter l'intersection entre le curseur et les modèles de véhicules, puis modifie temporairement leur `targetNode`

c. Statistiques en Temps Réel :

Vehicle Count :

Affichage : Compteur dynamique indiquant le nombre exact de véhicules actuellement instanciés et actifs sur la carte

Utilité : Permet de vérifier en direct si le flux reste conforme aux paramètres définis dans le menu de configuration et de détecter d'éventuelles anomalies (véhicules bloqués, fuites mémoire)

⇒ **Implémentation technique :** Cette valeur est récupérée via `simulation.getVehicleCount()` qui retourne la taille du `std::vector<std::unique_ptr<Vehicle>>`.

4.6 MENUS SUPERPOSÉS (OVERLAYS) :

a. Écran de Chargement :

Affiché lors de la transition entre la configuration et la simulation active, cet overlay informe l'utilisateur de la progression des opérations lourdes.

Éléments visuels :

- **Texte "CHARGEMENT DE LA VILLE…" :** Indication claire de l'étape en cours
- **Barre de progression animée :** Visualisation de l'avancement de la génération procédurale du `RoadGraph` et de l'instanciation des véhicules

⇒ **Implémentation technique :** Ce système utilise un thread secondaire pour éviter le gel de l'interface durant les calculs intensifs (génération de courbes de Bézier pour les routes, calcul des matrices de connexion du graphe).

b. Menu Pause et Paramètres Dynamiques :

Accessible à tout moment via la touche [P], ce menu permet des ajustements en temps réel sans interrompre définitivement la simulation.

Fonctionnalités :

- **Ajustement de vitesse :** Modification du time scale en plein déroulement pour ralentir ou accélérer l'observation.
- **Gestion dynamique de la flotte :** Ajout ou retrait de véhicules à chaud pour tester la résilience des intersections face à des pics de trafic.
- **Statistiques étendues :** Affichage de métriques avancées (FPS, temps de calcul par frame, densité par zone).

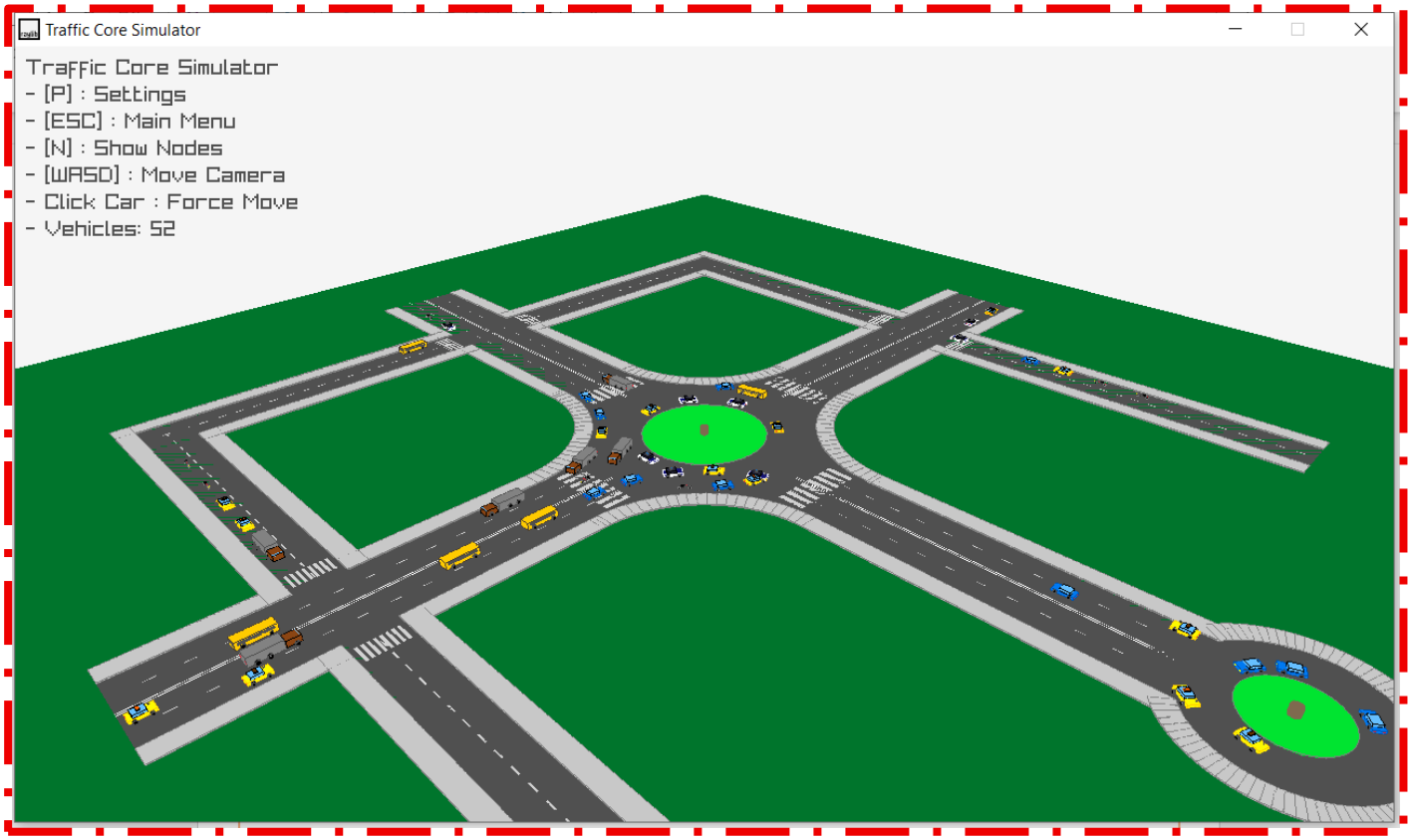
Contrôles :

- **Bouton [APPLY & RESUME] :** Applique instantanément les nouveaux réglages et reprend la simulation sans réinitialisation
- **Bouton [CANCEL] :** Annule les modifications et restaure les paramètres précédents

⇒ **Implémentation technique :** Ce menu utilise le pattern State avec une pause du deltaTime (dt = 0) dans Simulation::update(), gelant tous les véhicules sans désallouer leur mémoire.

4.7 PRINCIPES DE CONCEPTION UI :

a. Architecture Modulaire :



L'interface suit une séparation stricte des responsabilités :

- **TrafficInterface :** Classe gérant l'affichage des HUD et overlays
- **MainMenu / ConfigMenu / InGameMenu :** États distincts du pattern State Machine
- **App :** Contrôleur principal orchestrant les transitions

Cette modularité permet d'ajouter de nouveaux écrans (ex : menu de statistiques post-simulation) sans modifier le noyau existant.

b. Feedback Visuel Continu :

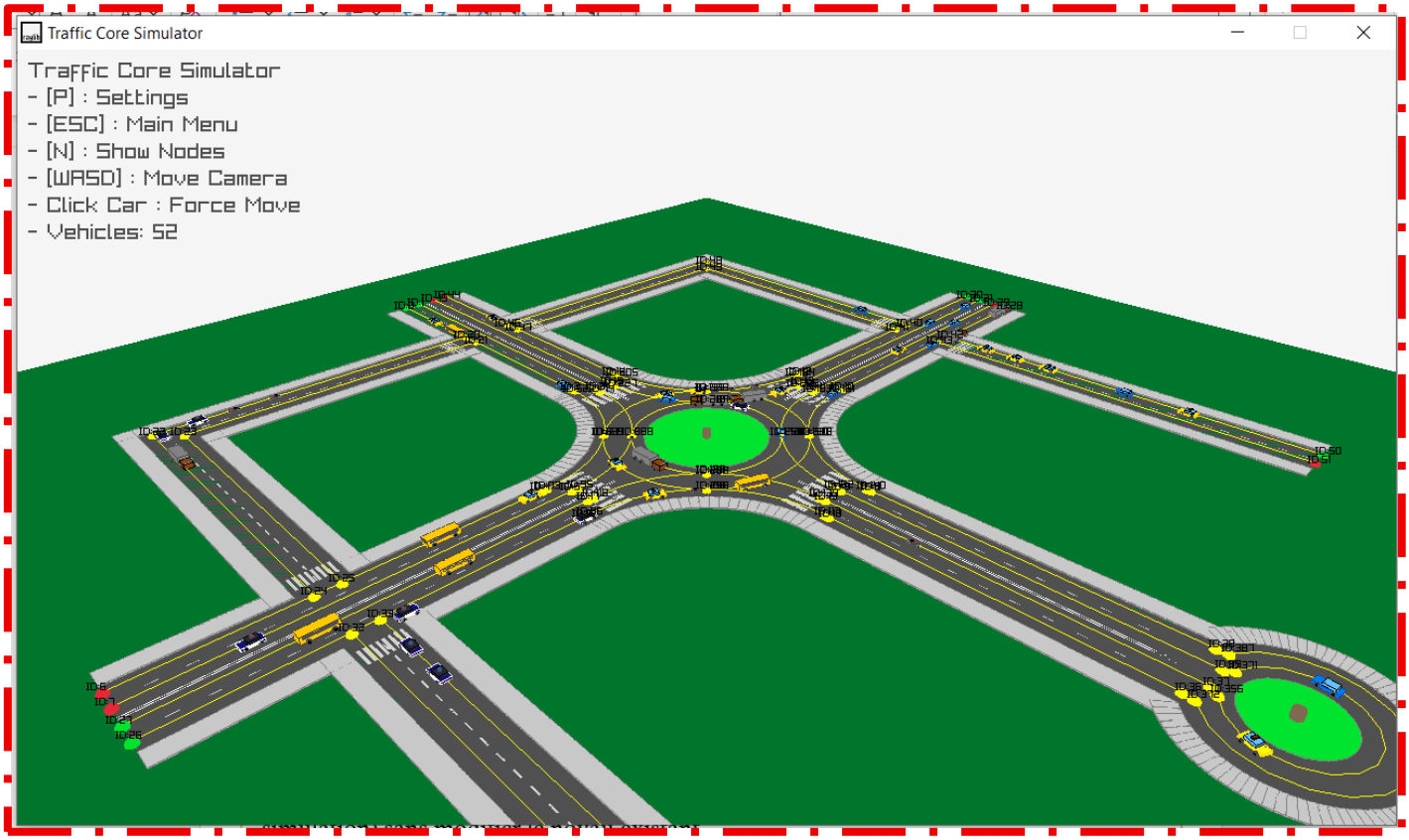
Chaque action utilisateur génère une réponse visuelle immédiate :

- ✓ Changement de couleur au survol des boutons ;
- ✓ Animation de pulsation lors de l'activation d'options ;
- ✓ Mise à jour en temps réel des compteurs et barres de progression ;

c. Accessibilité et Ergonomie :

- **Typographie :** Police sans-serif haute lisibilité avec contraste élevé
- **Codes couleur :** Rouge pour les alertes (limite atteinte), vert pour les états normaux, orange pour les avertissements
- **Positionnement :** Éléments critiques (statistiques, contrôles) toujours visibles dans la périphérie visuelle

L’interface avec l’affichage des nœuds :



5. Perspectives et Améliorations Futures : Vers un Écosystème Connecté :

Notre projet actuel, **Traffic Core**, constitue la fondation de l'infrastructure intelligente. Il ne se limite pas à la collecte de données ; il agit comme le générateur de vérité terrain ("Ground Truth") pour l'ensemble du système.

L'avenir de ce projet réside dans son intégration horizontale avec les trois autres sous-projets. L'objectif est de passer d'un système de **monitoring passif** à un écosystème de **gestion active et collaborative**.

5.1. Synergie avec le Sous-Projet 2 : ***Adaptive Signal Control***

Le lien entre le ***Traffic Core*** et le contrôle adaptatif des feux est le plus critique. Actuellement, notre système détecte la densité.

- **La Collaboration :** Le ***Traffic Core*** fournira les métriques de densité et de longueur de file d'attente en temps réel au module *Adaptive Signal Control*.
- **L'Impact :** Au lieu d'utiliser des minuteries fixes, les feux passeront au vert dynamiquement dès que notre système détectera une saturation sur une voie spécifique, ou créera une "onde verte" pour les services d'urgence identifiés par notre module de classification.
- **Flux de données :** ***Traffic Core*** (Détection Congestion) $\rightarrow$ API $\rightarrow$ Signal Control (Extension durée feu vert).

5.2. Intégration avec le Sous-Projet 3 : ***Routage Dynamique***

Le routage dynamique a besoin d'une "carte de chaleur" des incidents pour fonctionner. C'est le rôle du *Traffic Core*.

- **La Collaboration :** Notre système excelle dans la détection d'anomalies (accidents, objets sur la voie, ralentissements soudains). Ces événements seront instantanément "poussés" vers le moteur de routage.
- **L'Impact :** Dès que le *Traffic Core* repère un blocage, le Sous-Projet 3 peut recalculer les itinéraires pour les utilisateurs en amont, délestant ainsi le trafic avant même que le bouchon ne se forme. C'est une gestion prédictive basée sur notre vision par ordinateur.
- **Concept clé :** Le *Traffic Core* diagnostique le problème, le *Routage Dynamique* administre le remède.

5.3. Passerelle vers le Sous-Projet 4 : ***Smart Parking & EV Charging***

Une part significative du trafic urbain (souvent estimée à 30%) est causée par des véhicules cherchant un stationnement ("Cruising for parking").

- **La Collaboration :** Le ***Traffic Core*** peut analyser les flux entrants vers les zones de stationnement et corréler la baisse de trafic sur les axes principaux avec l'augmentation de la demande de parking.
- **L'Impact :** En croisant nos données de comptage de véhicules entrants avec les capteurs du Sous-Projet 4, nous pouvons prédire la saturation des parkings et guider les véhicules électriques (EV) vers des bornes libres *avant* qu'ils n'entrent dans une zone saturée.
- **Vision :** Une gestion holistique du voyage, de l'autoroute (Core) jusqu'à la place de stationnement (Smart Parking).

5.4. Architecture de Collaboration Technique

Pour matérialiser cette union, nous envisageons une architecture orientée événements :

- **Hub de Données Centralisé :** Toutes les données traitées par notre *Traffic Core* seront déversées dans un bus de messages partagé (ex: Kafka ou MQTT).
- **Abonnement aux Topics :**
  - Le SP2 s'abonne au topic traffic/density/high.
  - Le SP3 s'abonne au topic traffic/incident/alert.
  - Le SP4 s'abonne au topic traffic/flow/parking_zone.

6. Conclusion :

Face à l'urbanisation croissante et à la complexité des flux routiers, la gestion intelligente du trafic n'est plus un luxe, mais une nécessité impérieuse pour les villes modernes. Ce projet, **Traffic Core**, a permis de répondre à ce défi en développant une solution robuste de surveillance et d'analyse basée sur l'intelligence artificielle et la vision par ordinateur.

Au terme de ce travail, nous avons réussi à implémenter une chaîne de traitement complète, allant de la détection précise des véhicules via **YOLOv8** à leur suivi en temps réel avec **ByteTrack**, tout en intégrant l'estimation de la vitesse et la classification. La réalisation de notre tableau de bord interactif a concrétisé cette approche technique, transformant des flux vidéo bruts en **données décisionnelles exploitables** (KPIs, alertes, statistiques).

Cependant, la portée de ce projet dépasse ses seules fonctionnalités intrinsèques. Comme nous l'avons démontré, **"Traffic Core"** ne doit pas être vu comme un module isolé, mais comme le **système nerveux central** d'une infrastructure plus vaste. En fournissant la "vérité terrain" nécessaire, il habilite les autres sous-projets — du contrôle adaptatif des feux au routage dynamique — à fonctionner avec pertinence et efficacité.

En définitive, nous avons posé les fondations techniques d'un système capable de réduire la congestion, d'améliorer la sécurité routière et d'optimiser l'expérience urbaine. Ce projet démontre que la combinaison de l'IA et de l'ingénierie collaborative est la clé pour transformer nos routes passives en **autoroutes intelligentes et connectées**.