UNIVERSITÉ ABDELMALEK ESSAÂDI

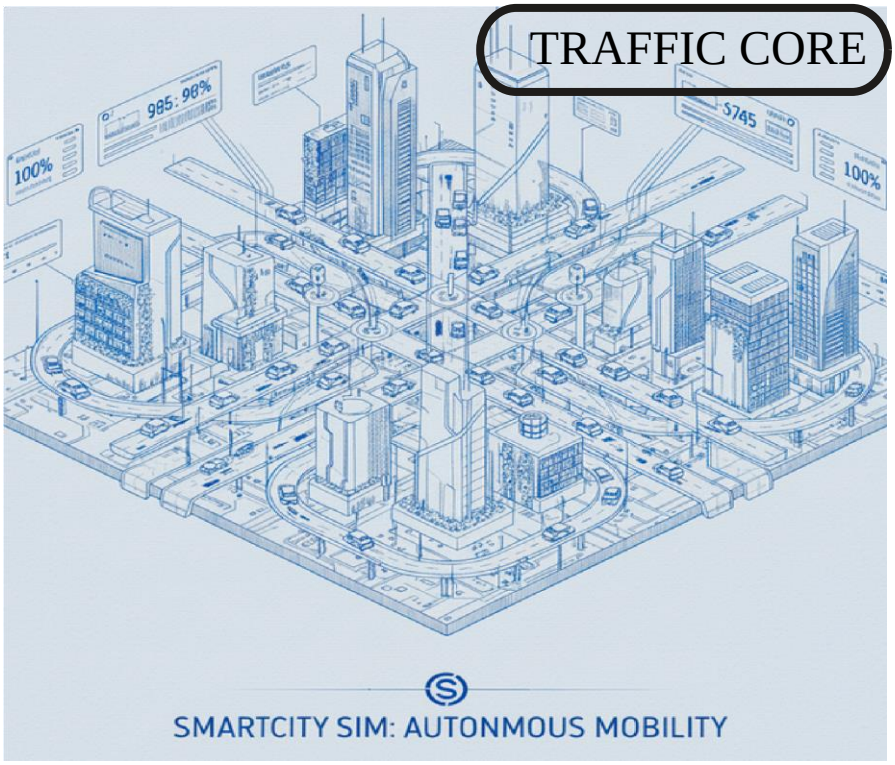
FACULTÉ DES SCIENCES ET TECHNIQUES DE TANGER

CYCLE LICENCE : IDAI

MODULE : PRGRAMMATION ORIENTÉ OBJET EN C++

RAPPORT DE PROJET

SMART-CITY AVEC C++ & RAYLIB



Encadré par :

Mme IKRAM BENABDELOUAHAB

Réalisé par :

OUARDA AIT EL FAKIH

FATIMA ZAHRAE EL BOUHSSINI

FATIMA BOUMEHAOUT

H ALIMA ACHABBAK

Année universitaire 2025-2026

Table des matières

Introduction générale

Les villes intelligentes (Smart Cities) représentent un modèle urbain moderne basé sur l'utilisation des technologies numériques afin d'optimiser la gestion des ressources, améliorer la mobilité urbaine et augmenter la qualité de vie des citoyens. Avec l'augmentation continue du nombre de véhicules en circulation, la gestion du trafic devient un enjeu majeur pour les grandes agglomérations.

Dans ce contexte, ce projet vise à concevoir et développer un **simulateur de Smart City** permettant de modéliser et d'analyser différents aspects de la mobilité urbaine. Le simulateur s'intéresse notamment à la circulation routière, à la gestion intelligente des feux de signalisation, au routage dynamique des véhicules, à la gestion des parkings et des véhicules électriques, ainsi qu'au traitement des véhicules prioritaires (urgences).

Le projet est structuré en **cinq sous-projets indépendants**, chacun correspondant à un module spécialisé. Cette organisation modulaire facilite la répartition du travail, la maintenance du code et l'évolution future du système. Le développement repose sur le langage **C++**, en appliquant rigoureusement les principes de la programmation orientée objet (POO), et utilise la bibliothèque graphique **Raylib** pour assurer une visualisation interactive de la simulation.

Ce rapport présente en détail les objectifs du projet, les choix technologiques et architecturaux, les principes de conception orientée objet adoptés, ainsi qu'une étude approfondie du sous-projet **Traffic Core**, qui constitue le cœur de la simulation.

1. Présentation générale du projet

1.1 Objectifs du projet

L'objectif principal du projet est de développer un simulateur capable de représenter de manière réaliste le trafic urbain dans une Smart City. Plus précisément, les objectifs sont les suivants :

- Concevoir une simulation de trafic flexible et paramétrable.
- Mettre en œuvre les concepts avancés de la programmation orientée objet en C++.
- Proposer une visualisation graphique interactive à l'aide de Raylib.
- Développer une architecture modulaire permettant l'intégration et l'interaction de plusieurs sous-systèmes.

1.2 Cahier des charges

Le projet doit respecter un ensemble de contraintes techniques et fonctionnelles imposées :

- Utilisation exclusive du langage **C++**.
- Application des principes fondamentaux de la POO : encapsulation, héritage et polymorphisme.
- Intégration de la bibliothèque **Raylib** pour la visualisation 2D ou 3D.
- Organisation claire du code source selon une architecture standard (include/, src/, tests/, demos/).
- Documentation du code à l'aide de commentaires et d'un fichier README.
- Mise en place d'un système de configuration externe, par exemple à l'aide de fichiers **JSON**, afin de paramétrer la simulation.
- Implémentation d'au moins cinq tests unitaires pour valider les classes principales.
- Utilisation d'au minimum deux design patterns par sous-projet.
- Fourniture d'un scénario de démonstration interactif.

1.3 Organisation en sous-projets

Le simulateur global est composé de cinq modules complémentaires :

- **Traffic Core** : cœur de la simulation de trafic.

- **Adaptive Signal** : gestion intelligente des feux de signalisation.
- **Dynamic Routing** : calcul dynamique des itinéraires.
- **Smart Parking & EV** : gestion des parkings et des véhicules électriques.
- **Emergency** : prise en charge des véhicules prioritaires.

Chaque module est développé de manière indépendante, tout en étant conçu pour interagir avec les autres modules dans le cadre d'un simulateur global.

2. Technologies et outils utilisés

 **C++** : langage principal du projet, choisi pour ses performances, sa flexibilité et son support de la programmation orientée objet.

 **Raylib** : bibliothèque graphique légère et performante utilisée pour la visualisation en temps réel de la simulation (routes, intersections, véhicules).

 **Visual Studio Code** : éditeur de code utilisé pour l'écriture, la compilation et le débogage du programme.

 **Git** : outil de gestion de versions permettant de suivre l'évolution du code source.

 **GitHub** : plateforme d'hébergement du projet facilitant la collaboration et la sauvegarde du code.

3. Architecture globale du système

3.1 Vue d'ensemble

L'architecture du simulateur repose sur une approche modulaire. Chaque sous-projet représente un composant indépendant possédant ses propres responsabilités, tout en communiquant avec les autres modules via des interfaces bien définies.

3.2 Organisation du code

Dossier include/ : Contient les fichiers d'en-tête (.h) définissant les **interfaces des classes principales** :

- Classes représentant les **nœuds** et **segments** du réseau.
- Classes pour les **intersections**.
- Classe de base **Vehicule** et classes dérivées (**Car**, **Bus**, **Truck**, etc.).
- Gestionnaires de trafic et de simulation.

Cette organisation favorise l'**encapsulation**, réduit le couplage et clarifie les responsabilités de chaque module.

Dossier src/ : Contient les fichiers source (.cpp) implémentant la **logique métier** :

- Déplacement dynamique des véhicules.
- Gestion du graphe routier et des intersections.
- Calcul des trajets et mise à jour des positions.

Chaque fichier source correspond à un fichier d'en-tête dans include/, assurant cohérence et clarté.

Dossier demos/ : Contient les scénarios de démonstration, tels que boukhalf_simulation.cpp, qui illustrent :

- Création du réseau routier.
- Ajout de véhicules.
- Lancement et observation de la simulation.

Ces fichiers servent à tester et valider le module Traffic Core.

Dossier assets/ : Contient les **ressources graphiques 3D**, notamment :

- Modèles de véhicules et bâtiments au format .glb.
- Textures et éléments visuels pour Raylib.

Ces ressources permettent un rendu réaliste et interactif.

Dossier tests/ : Contient les **tests unitaires** visant à valider le fonctionnement des principales classes :

- Vérification du comportement des véhicules.
- Tests sur la construction et la navigation dans le réseau routier.
- Validation des algorithmes de calcul de trajets.

Les tests unitaires assurent la fiabilité, facilitent la maintenance et permettent de détecter rapidement les erreurs lors des évolutions du code. Leur intégration reflète une démarche professionnelle de développement logiciel.

Dossier build/ : Généré automatiquement par CMake, il contient les fichiers de compilation et les exécutables. Sa séparation du code source garantit une organisation propre et modulable.

Fichier CMakeLists.txt

Gère la compilation et les dépendances, notamment Raylib, et configure le projet pour différents environnements.

Fichier README.md : Documente le projet : objectifs, instructions de compilation, organisation du code et fonctionnalités principales.

4. Conception orientée objet

4.1 Diagramme de classe

Voici les classes qui structurent le code :

- **TrafficManager :** Le "cerveau" de la simulation. Gère la liste de tous les véhicules et les mises à jour globales.
- **RoadNetwork :** Contient le graphe du réseau (Noeuds et Segments). Contient la logique A*.
- **Node / Intersection :** Représentent les carrefours ou points de connexion.
- **RoadSegment:** Représente une route reliant deux nœuds.
- **Vehicule (Abstrait) :** Classe de base gérant la physique (IDM), la position, et l'itinéraire.
- **Car(Hérite de Vehicule) :** Implémentation concrète, ajoute la gestion du carburant, le type de modèle 3D spécifique, et les caractéristiques (vitesse max, couleur).

- **CarFactory :** Implémente le Factory Pattern. Centralise la création complexe des voitures (choix du modèle aléatoire, configuration).
- **ModelManager :** Implémente le Singleton Pattern. Charge les modèles 3D une seule fois en mémoire et les distribue pour optimiser les performances

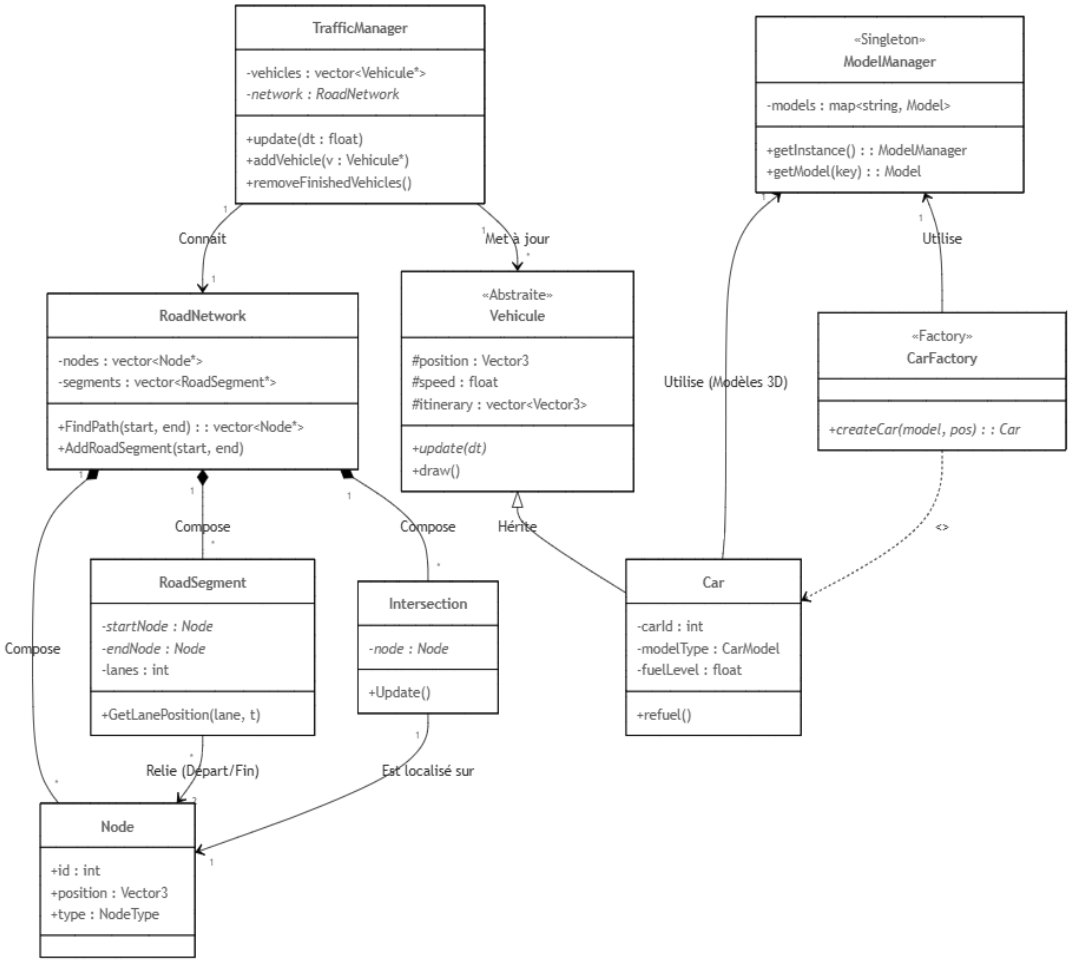


Figure 1 : diagramme de classe de la simulation

4.2 Principes POO appliqués

Encapsulation : Toutes les données membres (ex: position, speed dans Vehicule) sont privées ou protégées. L'accès se fait via des méthodes publiques (getPosition(), update()), garantissant l'intégrité des données.

Héritage : La classe Vehicule factorise le code commun (physique, mouvement). Les classes enfants (Car , Bus) héritent de ces capacités tout en pouvant ajouter des spécificités (modèles 3D différents, tailles variées).

Polymorphisme :

- La méthode virtual void update(float dt) permet de traiter tous les véhicules de manière uniforme dans la boucle principale, quel que soit leur type réel.
- Le conteneur std::vector<std::unique_ptr<Vehicule>> stocke indifféremment des voitures, bus ou camions

5. Design Patterns Utilisés

L'architecture intègre plusieurs patrons de conception pour résoudre des problèmes classiques :

5.1 Factory Pattern (Fabrique)

Problème : La création de véhicules nécessite une logique complexe (choix du modèle 3D, configuration des performances, positionnement).

Solution : La classe VehiculeFactory (et CarFactory) centralise cette logique.

Avantage : Le code client (la simulation) demande simplement "Une voiture à telle position" sans connaître les détails d'instanciation (new Car(...), chargement des assets...). Cela respecte le principe Open/Closed : ajouter un véhicule ne casse pas le code existant.

5.2 Singleton

Problème : Les modèles 3D (meshes géométriques) sont lourds et ne doivent être chargés qu'une seule fois en mémoire, mais accessibles partout.

Solution : La classe ModelManager est un Singleton. Elle garantit qu'une seule instance du gestionnaire de ressources existe.

Avantage : Optimisation drastique de la mémoire et accès global facilité aux ressources (ModelManager::getInstance()).

6. Analyse du Sous-Projet : Traffic Core

6.1 Modélisation du Réseau Routier

Le réseau est un Graphe Orienté pondéré :

- **Noeuds (Node) :** Représentent les intersections ou les points de jonction (ronds-points). Ils possèdent un type (SIMPLE_INTERSECTION, ROUNDABOUT, TRAFFIC_LIGHT).
- **Arêtes (RoadSegment) :** Relient deux noeuds. Elles possèdent des propriétés physiques (longueur, courbure) et logiques (nombre de voies, sens de circulation).
- La structure permet de modéliser des routes à sens unique ou double sens (via deux segments inverses invisibles).

6.2 Modélisation des Véhicules

Chaque véhicule est un agent autonome :

- **Physique :** Accélération, freinage, vitesse max.
- **Navigation :** Capacité à suivre une liste de points (itinéraire).
- **Sécurité :** Détection de collision frontale (freinage d'urgence si un obstacle est détecté devant).

6.3 Algorithme de Routage (Pathfinding)

Pour qu'un véhicule aille d'un point A à un point B, nous avons implémenté l'algorithme A (A-Star)* dans RoadNetwork::FindPath.

- **Fonctionnement :** Il explore le graphe en privilégiant les chemins qui se rapprochent de la destination (grâce à une heuristique de distance euclidienne).
- **Performance :** Plus efficace que Dijkstra pour ce type de problèmes spatiaux, garantissant le chemin le plus court de manière optimale.

6.4 Gestion de la Simulation

La classe TrafficManager agit comme le moteur de la simulation :

- **Boucle Update :** À chaque image (frame), elle met à jour la position et l'état de tous les véhicules.
- **Cycle de vie :** Elle supprime les véhicules arrivés à destination et permet l'injection dynamique de nouveaux véhicules pour maintenir la densité du trafic.

7. Captures d'écran de la simulation

Visualisation et Caméra

Le rendu utilise Raylib en mode 3D. Un système de caméra avancé a été développé avec 3 modes :

- **Orbital :** Pour une vue tactique (type RTS).
- **Free Fly :** Pour explorer la ville (type FPS).

Figure 2 : informations sur la simulation

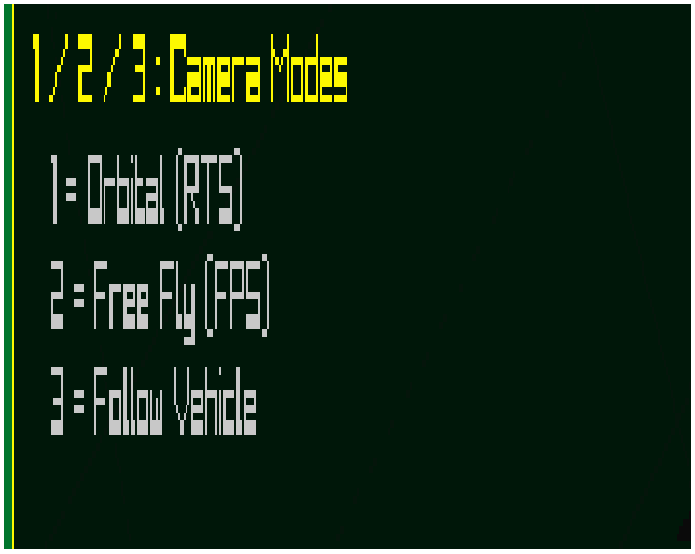


Figure 3 : les modes de caméra

- **Suivi :** Pour s'attacher à un véhicule spécifique et voir son point de vue.

Figure 4 : suivi le chemin du véhicule en cliquant sur le bouton 3

Simulation de Trafic Dense

- Gestion fluide de centaines de véhicules simultanés (Voitures, Bus, Camions).
- Le système assure un spawn automatique pour maintenir la ville active.

Figure 5 : Simulation de trafic dense avec gestion des collisions

Infrastructure Routière Complexe

- Réseau fonctionnel comprenant des routes à plusieurs voies.
- Gestion réaliste d'un rond-point central connectant plusieurs axes.
- Support des sens de circulation (y compris double sens).

Figure 6 : Architecture du réseau routier et rond-point

Environnement 3D Immersif

- Intégration de modèles 3D (Bâtiments, Stade, Végétation, Fontaines).
- Environnement avec ciel et éclairage (Skybox).

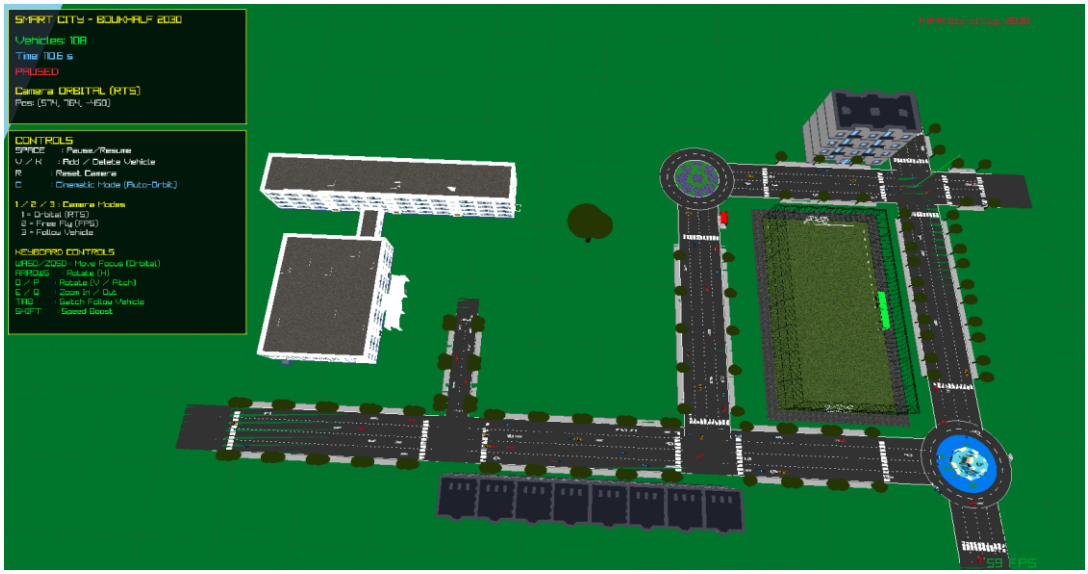


Figure 7 : Environnement 3D (Bâtiments et Végétation)

Interaction Utilisateur Temps Réel

- Pause/Reprise de la simulation.
- Ajout manuel de véhicules (Touche V).
- Suppression de véhicules (Touche K).
- Bascule de mode cinématique (Touche C).

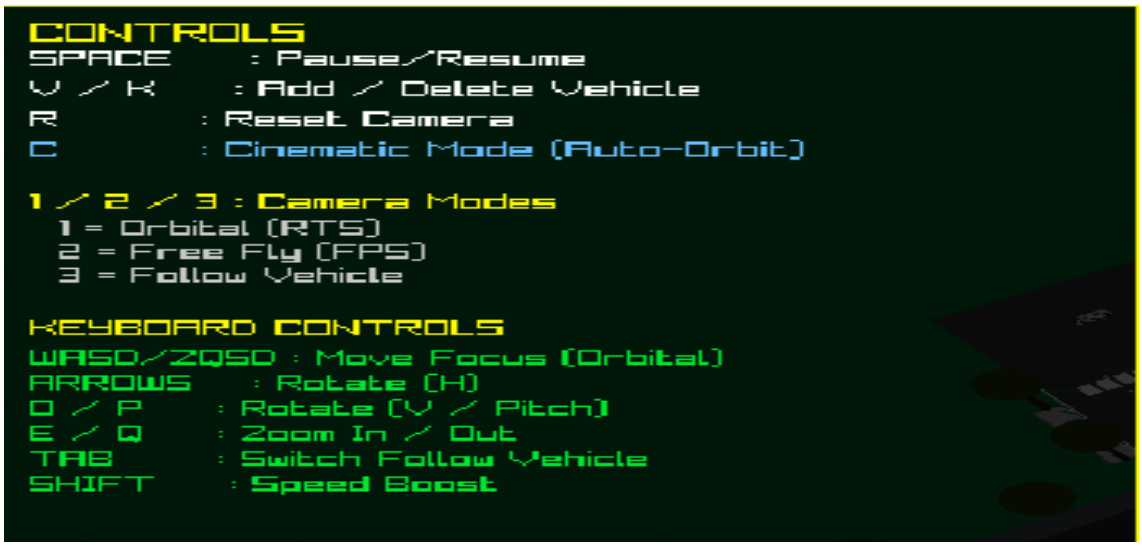


Figure 8 : Interface utilisateur et contrôle de la simulation

Points Forts

- **Architecture propre :** Le code est modulaire et facile à étendre.
- **Rendu Visuel :** L'utilisation de modèles 3D et d'un éclairage basique rend la simulation immersive.
- **Robustesse :** L'utilisation de pointeurs intelligents (std::unique_ptr) évite les fuites de mémoire.

Conclusion

Le développement du module **Traffic Core** pour ce simulateur de Smart City a permis de concrétiser une approche rigoureuse et professionnelle de la programmation orientée objet en C++. Le projet a réussi à modéliser un **réseau routier réaliste**, à gérer le comportement de différents types de véhicules et à implémenter des **algorithmes de calcul de trajets** efficaces.

L'organisation modulaire du code, avec des dossiers clairement définis (include, src, tests, demos, assets et build), permet une **maintenabilité optimale** et facilite l'ajout futur de nouveaux modules ou fonctionnalités. La présence de **tests unitaires** renforce la fiabilité du système et reflète les bonnes pratiques de l'ingénierie logicielle professionnelle.

Grâce à l'intégration de **Raylib**, la simulation bénéficie d'une interface graphique interactive, rendant l'observation des flux de trafic intuitive et pédagogique. Ce projet constitue une **base solide pour un simulateur urbain global**, pouvant être enrichi par l'ajout de modules avancés tels que la gestion intelligente des feux, les véhicules d'urgence ou l'analyse environnementale.

En perspective, ce simulateur ouvre la voie à des études avancées sur le comportement du trafic, l'optimisation des infrastructures et l'intégration de systèmes intelligents pour les villes de demain. Il offre un outil concret et modulable pour expérimenter, analyser et anticiper les problématiques urbaines, répondant ainsi pleinement aux objectifs pédagogiques et technologiques fixés.

Références

Raylib : https://www.raylib.com/index.html

Poly pizza : https://poly.pizza/

Sketchfab: https://sketchfab.com/search?q=car&type=models

Free3D : https://free3d.com/3d-models/car