**Université Abdelmalek Essaâdi**
**Faculté des sciences et techniques de Tanger**
جامعة عبد المالك السعدي
كلية العلوم والتقنيات

## RAPPORT DE PROJET DE FIN DE MODULE

Filière : Licence IDAI

---

# Conception et réalisation d'un système de routage dynamique pour une Smart City, intégrant la simulation du trafic, la gestion d'événements et le recalcul d'itinéraires en temps réel.

---

## PROJET DE FIN DE MODULE : SIMULATEUR D'UNE SMART CITY EN C++ SOUS-PROJET 3 : Routage dynamique

**Encadré par :**

Pr. Ikram BEN ABDEL OUAHAB

**Réalisé par :**

ZIOUANI Doha
OUTAHAR Doha
ELHADI Ahlam
GRAOUI Nawras

**Année universitaire : 2025 – 2026**

# Résumé

Ce sous-projet s'inscrit dans le cadre du projet *Smart City* et vise à concevoir une simulation 2D de **routage dynamique** permettant le **recalcul automatique d'itinéraires** lorsque des perturbations surviennent (accident, congestion, fermeture de route, urgence).

Le réseau routier est modélisé sous forme de **graphe orienté** (nœuds/intersections et routes/arêtes) et la planification de trajet repose sur un planificateur **interchangeable** basé sur le pattern **Strategy**, intégrant **A*** (par défaut) et **Dijkstra** (référence) afin de comparer les comportements. Les événements, créés via un pattern **Factory**, modifient l'état ou le coût des routes (sévérité, durée) et déclenchent un reroutage ciblé des véhicules concernés, tout en tenant compte du trafic afin de limiter les effets de reroutage massif. La simulation est paramétrable via un fichier **JSON** (réseau, véhicules, événements, mode Normal/Dynamique) et s'appuie sur **Raylib** pour visualiser en temps réel le réseau, les véhicules, les routes affectées et les indicateurs de simulation. Enfin, une stratégie de **tests unitaires** (minimum 5) permet de valider les classes principales et de garantir la robustesse du système.

# Table des matières

# Liste des figures

# Glossaire

**Accident (événement)** : incident simulé qui perturbe la circulation (ralentissement fort ou blocage d'une route).

**Algorithme de plus court chemin** : méthode qui calcule le meilleur trajet dans un graphe selon un coût (temps, distance, etc.).

**A*** : algorithme de recherche de chemin qui utilise une heuristique pour accélérer le calcul (souvent plus rapide que Dijkstra).

**Arête (Edge)** : route dans le graphe reliant deux intersections (nœuds).

**Blocage de route** : état d'une route devenue impraticable (coût infini ou route désactivée).

**Capacité d'une route** : nombre maximal de véhicules qu'une route peut supporter avant congestion.

**Congestion** : surcharge de trafic qui augmente le temps de trajet sur une route.

**Coût (Weight)** : valeur associée à une route (ex. temps estimé) utilisée par l'algorithme pour comparer les itinéraires.

**Densité du trafic** : quantité de véhicules présents sur une route à un instant donné.

**Dijkstra** : algorithme classique qui trouve le chemin optimal dans un graphe pondéré (sans heuristique).

**Événement** : perturbation du réseau (accident, fermeture, embouteillage) qui modifie les coûts ou l'état des routes.

**Graphe** : modèle mathématique représentant le réseau routier par des nœuds et des arêtes.

**Heuristique** : estimation utilisée par A* pour guider la recherche vers la destination (ex.

distance à vol d'oiseau).

**Intersection** : point où plusieurs routes se croisent ; correspond à un nœud du graphe.

**Itinéraire** : suite ordonnée de routes (ou de nœuds) à suivre pour aller d'un point A à un point B.

**Mise à jour (Update)** : calcul régulier de l'état de la simulation (positions, densité, coûts, événements).

**Nœud (Node)** : élément du graphe représentant une intersection ou un point du réseau.

**Optimisation de trajet** : choix du meilleur itinéraire selon un critère (souvent le temps minimal plutôt que la distance).

**Poids dynamique** : coût d'une route qui change dans le temps selon trafic et événements.

**Recalcul d'itinéraire** : nouvelle planification de route effectuée pendant la simulation.

**Reroutage (Routage dynamique)** : action de changer automatiquement d'itinéraire en réponse à une congestion ou un incident.

**Seuil de reroutage** : condition minimale (gain de temps) pour autoriser un reroutage et éviter les changements inutiles.

**Simulation temps réel** : simulation qui évolue en continu avec un rythme proche du temps réel (frames/ticks).

**Smart City** : ville "intelligente" utilisant des systèmes numériques pour mieux gérer mobilité, trafic et services.

**Tick / Frame** : unité de temps de la simulation (à chaque tick, on met à jour le système).

**Temps estimé de traversée** : durée approximative pour parcourir une route (base + pénalités de trafic/événements).

**Visualisation (Raylib)** : affichage graphique de la simulation via la bibliothèque Raylib.

# Chapitre 1

# Introduction

### 1.0.1 Contexte général du projet : Smart City

Le projet *Smart City* vise à exploiter des outils numériques pour améliorer la gestion de la mobilité urbaine. Dans une ville, l'état du trafic peut évoluer rapidement à cause de congestions, d'accidents ou de fermetures temporaires, ce qui impacte directement les temps de trajet. Dans ce contexte, il devient nécessaire de disposer de mécanismes capables d'analyser l'état du réseau routier et d'adapter les déplacements en conséquence.

### 1.0.2 Objectif du Sous-Projet 3 : Routage dynamique

L'objectif du sous-projet *Routage Dynamique* est de concevoir une simulation interactive permettant de calculer un itinéraire optimal entre une source et une destination, puis de **recalculer automatiquement** cet itinéraire lorsque l'état du réseau change. Le système s'appuie sur un graphe routier (nœuds et routes) et sur des algorithmes de recherche de chemin (A* et Dijkstra) afin de comparer le comportement en **mode Normal** (itinéraire fixe) et en **mode Dynamique** (reroutage en temps réel).

### 1.0.3 Problématique

La problématique principale est la suivante : *comment adapter en temps réel les trajets des véhicules lorsqu'un événement (accident, congestion, route bloquée, urgence) modifie le coût ou l'accessibilité de certaines routes, tout en évitant des recalculs inutiles et en gardant une simulation fluide ?* Pour répondre à cette problématique, le projet propose un modèle d'événements affectant les routes, une mise à jour régulière des coûts (poids dynamiques) et un mécanisme de reroutage ciblé pour les véhicules réellement impactés.

**Un défi particulier** réside dans la gestion des effets de bord : le reroutage simultané d'un grand nombre de véhicules peut entraîner la saturation des chemins alternatifs. Il est par conséquent impératif de concevoir une stratégie de reroutage qui favorise la diversification des chemins et pondère les coûts en fonction de la charge actuelle du trafic.

### 1.0.4 Les Outils utilisés


**Raylib** est une bibliothèque graphique open-source en C, conçue pour créer rapidement des applications et des simulations 2D/3D. Elle est légère, simple à prendre en main et permet d'afficher facilement la carte, les véhicules, les événements et l'interface de contrôle.

**CMake** est un outil multiplateforme qui génère des fichiers de build (Makefiles, Visual Studio, etc.) à partir du fichier `CMakeLists.txt`. Il automatise la configuration, la compilation et la gestion des dépendances du projet C/C++.


**MinGW-w64** est une distribution du compilateur **GCC** pour Windows. Elle fournit la chaîne d'outils (compilateur, assembleur, éditeur de liens) nécessaire pour construire et exécuter des programmes C/C++ sur Windows, et s'intègre facilement avec CMake pour produire l'exécutable du projet.

**Visual Studio Code** est un éditeur de code extensible (Windows, Linux, macOS). Il facilite le développement grâce aux extensions C/C++, au débogage, à l'exécution de tâches et à une navigation rapide dans les fichiers du projet.

**Zed Editor** est un éditeur moderne et performant, pensé pour une expérience fluide et minimaliste. Il met l'accent sur la rapidité et la productivité, et peut être utilisé comme alternative légère pour éditer et organiser le code source.

**Git** est un système de gestion de versions distribué. Il permet de suivre l'historique du code, gérer les branches, revenir à des versions stables et faciliter le travail en équipe.


**GitHub** est une plateforme basée sur Git pour héberger les dépôts, collaborer (issues, pull requests), partager le code et centraliser la documentation du projet.

# Chapitre 2

# Méthodologie et rôle individuel

## 2.1 Approche méthodologique adoptée au sein de l'équipe

Dans le cadre du projet *Smart City* (Sous-Projet 3 : *Routage Dynamique*), l'équipe a adopté une démarche de développement itérative, structurée autour de livraisons progressives. Cette approche a permis de valider rapidement une version fonctionnelle du simulateur, puis d'améliorer progressivement le module de recalcul dynamique des itinéraires (prise en compte des événements, congestion, reroutage en temps réel, puis visualisation).

### Planification et répartition des tâches

Les travaux ont été organisés à partir des exigences du cahier des charges : simulation paramétrable, planificateur de trajets (A* / Dijkstra), stratégie anti-blocage, intégration Raylib, tests unitaires et configuration JSON. Les tâches ont ensuite été réparties selon les responsabilités : modélisation du graphe routier (nœuds/routes), implémentation des algorithmes de recherche de chemin, gestion des événements (accident, fermeture, congestion), orchestration de la simulation et développement de l'interface graphique.

### Développement modulaire et intégration progressive

Afin d'assurer la maintenabilité et de faciliter l'intégration, l'équipe a privilégié une architecture modulaire séparant clairement les responsabilités (Graph, Vehicle, Event, PathPlanner, Simulation, Renderer). Le projet a été structuré en répertoires dédiés (`include/`, `src/`, `tests/`, `demos/`), ce qui a simplifié l'évolution du code et l'ajout de fonctionnalités sans impacter l'ensemble du système.

### Choix techniques guidés par la conception

La conception a été consolidée par l'utilisation de patrons de conception exigés et adaptés au besoin : le pattern *Strategy* pour permettre le changement d'algorithme (A* ou Dijkstra) au sein du planificateur, et le pattern *Factory* pour centraliser la création des véhicules et des

événements. Cette organisation a renforcé la flexibilité du module et a facilité les comparaisons de comportements en simulation.

### Validation (tests) et démonstration

La validation a été effectuée en continu à l'aide de tests unitaires sous *Catch2* afin de sécuriser les composants critiques (cohérence du graphe, calcul de chemin, pondération par le temps, application/levée des événements, détection du besoin de reroutage). En parallèle, une démo interactive sous Raylib a servi de preuve fonctionnelle : déclenchement d'un incident (ex. touche `SPACE`), bascule entre mode normal et dynamique (ex. touche `R`) et observation du reroutage ainsi que de l'impact sur la fluidité.

## 2.2 Positionnement de notre rôle et responsabilités principales

Dans le cadre du projet *Simulateur d'une Smart City en C++ (POO & Raylib)* et plus précisément du *Sous-projet 3 : Routage dynamique*, nous avons assuré la conception et l'implémentation d'un module de recalcul d'itinéraires capable de s'adapter aux perturbations (accidents, congestions, fermetures de routes) en temps réel. Notre positionnement a été à la fois **technique** (algorithmes et modélisation) et **intégration** (orchestration de la simulation et visualisation), afin de garantir une solution robuste, maintenable et démontrable.

### Responsabilités principales

Nos responsabilités se sont articulées autour des axes suivants :

- **Modélisation du réseau routier (graphe)** : structuration du graphe orienté (nœuds/intersections et routes/arêtes) et définition des attributs nécessaires au calcul des coûts (longueur, vitesse maximale, capacité, état utilisable/bloqué).
- **Planification de chemin et optimisation** : mise en place du planificateur de trajets (*PathPlanner*) basé sur une stratégie interchangeable (*Strategy*) permettant d'utiliser A* ou Dijkstra, avec un objectif prioritaire : **minimiser le temps de trajet** plutôt que la distance.:contentReference
- **Gestion des événements** : conception/prise en charge des perturbations (ACCIDENT, TRAFFIC_JAM, ROAD_CLOSURE, etc.) et application de leur impact sur les routes via sévérité et durée, afin de refléter les conditions réelles de circulation.
- **Reroutage dynamique en temps réel** : intégration du mécanisme de reroutage orchestré par la classe *Simulation* (détection d'impact, notification, recalcul via *PathPlanner*) en ciblant uniquement les véhicules concernés, pour garantir une réactivité optimale.

- **Paramétrage et reproductibilité** : mise en place de la configuration *JSON* pour charger le graphe, les véhicules initiaux et les paramètres, permettant de créer de nouveaux scénarios sans recompilation.
- **Tests et validation** : réalisation des tests unitaires avec *Catch2* sur les classes critiques (Graph, PathPlanner, Event, Vehicle) ainsi qu'un test d'intégration pour valider l'interaction Simulation–PathPlanner en cas d'événement bloquant.
- **Démonstration et visualisation** : préparation de la démo interactive sous Raylib (déclenchement d'événements et bascule Normal/Dynamique) afin d'illustrer clairement l'intérêt du reroutage en conditions perturbées.

## Résultat attendu

À l'issue de nos contributions, le système fournit une simulation visuelle cohérente où les véhicules calculent un itinéraire initial, puis adaptent leur trajectoire lorsque les coûts changent ou lorsqu'une route est bloquée, tout en restant conforme aux exigences de modularité, de maintenabilité et de démonstration du projet.

# Chapitre 3

# Conception et Développement

## 3.1 Conception et Développement du sous projet - Routage Dynamique

### 3.1.1 Objectif et principes de conception

Notre sous-projet vise à mettre en place un **système de recalcul dynamique des itinéraires** dans une Smart City, afin que les véhicules s'adaptent en temps réel aux perturbations du réseau (congestion, accidents, routes bloquées, urgences). Contrairement à un routage statique, l'objectif n'est pas uniquement de trouver le chemin le plus court, mais **le plus rapide** en intégrant l'état courant de la circulation. Une attention particulière est portée au cas où un reroutage massif pourrait créer une nouvelle congestion, ce qui impose une stratégie de recalcul réellement adaptée.

### 3.1.2 Modélisation (graphe routier, véhicules, événements)

Le réseau routier est modélisé par un **graphe orienté** :

- **Nœuds (Node)** : représentent les intersections, définis par des coordonnées spatiales (x, y).
- **Routes (Route)** : représentent les arêtes reliant deux nœuds et portent les attributs nécessaires au calcul du coût : longueur, vitesse maximale, capacité, niveau de trafic actuel et état opérationnel (utilisable ou bloqué).

**Les véhicules** sont des entités autonomes possédant un identifiant, une position, une destination, une vitesse/type, et un **chemin planifié** (séquence de nœuds), ainsi qu'un mécanisme interne permettant de détecter le besoin de reroutage.

**Les événements** modélisent des perturbations temporaires appliquées à une route (ACCIDENT, TRAFFIC_JAM, ROAD_CLOSURE, EMERGENCY) et sont caractérisés par un type, une route affectée, une sévérité (0.0 à 1.0) et une durée.

### 3.1.3 Architecture logicielle et organisation des modules

Le système est structuré selon une **architecture modulaire en couches** favorisant la séparation des responsabilités :

- **Renderer (Raylib)** : couche de visualisation 2D, responsable de l'affichage des routes, véhicules et perturbations.
- **Simulation** : orchestrateur central gérant le temps, la mise à jour des entités, le reroutage et les statistiques.
- **Graph / Vehicles / Events** : noyau métier représentant le réseau, la flotte et les perturbations.
- **PathPlanner** : module de planification utilisant le *pattern Strategy* pour choisir dynamiquement l'algorithme (A* ou Dijkstra).

### 3.1.4 Conception POO et Design Patterns

Deux patrons de conception ont été intégrés :

- **Factory** : **VehicleFactory** et **EventFactory** centralisent la création des véhicules et des événements (aléatoires ou configurés), ce qui simplifie l'extension (ajout de nouveaux types) et garantit une création cohérente.
- **Strategy** : appliqué au **PathPlanner** via l'interface *PathfindingStrategy* et ses implémentations (*AStarStrategy*, *DijkstraStrategy*), afin de changer l'algorithme au moment de l'exécution.

**Prise en compte des routes bloquées :** en cas de fermeture (*ROAD_CLOSURE*), la route est considérée **inutilisable** et est exclue du calcul (ou reçoit un coût très élevé), empêchant sa sélection par l'algorithme.

**Effet recherché (diversification du trafic) :** afin d'éviter qu'un reroutage massif ne crée un *nouveau point de congestion*, le coût est **ajusté dynamiquement** en fonction de la charge. Ainsi, une route déjà très utilisée devient plus coûteuse, ce qui incite le planificateur à répartir les véhicules sur plusieurs alternatives et améliore la fluidité globale.

#### Implémentation du routage : fonction de coût et congestion

Dans notre simulation, le calcul d'itinéraire repose sur un graphe routier où chaque route possède des attributs (longueur, vitesse maximale, capacité, état et trafic actuel). Le planificateur utilise ces informations pour évaluer un **coût dynamique** représentant un **temps de trajet estimé**. Ainsi, le routage ne se limite pas à minimiser la distance : il vise à sélectionner un itinéraire plus rapide en fonction des conditions courantes.

**Prise en compte du trafic (congestion) :** le coût d'une route augmente lorsque le trafic s'intensifie. Une route plus fréquentée devient donc moins attractive, ce qui incite naturellement

le système à répartir les véhicules sur des alternatives disponibles et à limiter la formation de nouveaux points de congestion après un reroutage.

**Impact des événements :** les événements par exemple :ACCIDENT,TRAFFIC JAM,ROAD CLOSURE, URGENCE modifient l'état de certaines routes. Selon le cas, une route peut être pénalisée (ralentissement) ou rendue indisponible (route bloquée). Ces changements se répercutent directement sur les coûts utilisés par le planificateur, ce qui justifie le recalcul d'itinéraires en mode dynamique.

**Objectif :** en ajustant continuellement les coûts à partir du trafic et des événements, le système évite de concentrer tous les véhicules sur un même détour et améliore la fluidité globale de la simulation, tout en restant cohérent avec l'état réel du réseau.

### 3.1.5 Validation, tests et démonstration Raylib

La validation repose sur des **tests unitaires Catch2** couvrant les classes critiques (Graph, PathPlanner, Event, Vehicle) et un test d'intégration vérifiant l'interaction Simulation–PathPlanner lors d'un événement bloquant.

### 3.1.6 Structure de Projet

**build** → Fichiers de compilation générés automatiquement par CMake (exécutable et artefacts).
**include** → Répertoire contenant les fichiers d'en-têtes (.h/.hpp) des modules du sous-projet (graphe, véhicules, événements, planification).
**src** → Répertoire contenant l'implémentation (.cpp) de la simulation, du routage dynamique et de la mise à jour du trafic.
**external(raylib)** → Répertoire contenant les bibliothèques externes nécessaires (ex. Raylib) et/ou leur configuration d'intégration.
**config** → Fichiers de configuration *JSON* (réseau, paramètres, véhicules, événements) pour rejouer des scénarios sans modifier le code.
**demos** → Scénarios de démonstration (mode normal vs mode dynamique, déclenchement d'événements en cours de simulation).
**tests** → Tests unitaires (Catch2) pour valider les classes critiques (Graph, PathPlanner, Event, Vehicle, Simulation).
**assets/** → Ressources graphiques/sonores utilisées pour la visualisation (textures, icônes, polices, etc.).
**CMakeLists.txt** → Fichier de configuration CMake (compilation, dépendances, options).
**README.md** → Documentation du projet (installation, compilation, exécution, contrôles, description).

**On sépare le sous-projet en plusieurs classes :**

**Simulation** : gère la boucle principale de la simulation, la mise à jour temporelle et l'orchestration du reroutage.
**Graph** : stocke le réseau routier sous forme de graphe (nœuds et routes) et permet le chargement depuis JSON.
**Node** : représente une intersection (identifiant et coordonnées).
**Route** : représente une route orientée (longueur, vitesse max, capacité, niveau de trafic, état bloqué/actif).
**Vehicle** : gère le déplacement des véhicules, leur itinéraire courant et la détection du besoin de reroutage.
**Event** : modélise les perturbations (accident, congestion, fermeture) et applique leur impact sur les routes (sévérité, durée).
**PathPlanner** : calcule le chemin optimal (objectif : temps de trajet minimal) et pilote l'algorithme choisi.
**AStarStrategy** : implémentation de l'algorithme A* (avec heuristique) pour accélérer la recherche de chemin.
**DijkstraStrategy** : implémentation de Dijkstra pour le calcul de plus court chemin (référence sans heuristique).
**Renderer** : assure l'affichage Raylib (routes, véhicules, événements, routes bloquées) et l'interaction de démonstration.
**VehicleFactory** : crée les véhicules selon leur type/paramètres (pattern Factory).
**EventFactory** : crée les événements à partir de la configuration/scénario (pattern Factory).

**Remarque :**
Les classes sont séparées en fichiers d'en-tête (.h/.hpp) et fichiers d'implémentation (.cpp). La compilation est gérée via CMake (C++17), et la visualisation est assurée par Raylib.

## 3.2 Implémentation

### 3.2.1 Algorithmes de routage (A* / Dijkstra) et recalcul en temps réel

Dans le sous-projet *Routage Dynamique*, la ville est modélisée sous forme de **graphe orienté** : les **nœuds** représentent les intersections et les **routes** (arêtes) représentent les segments de circulation. Chaque route possède un **coût dynamique** mis à jour selon l'état du trafic (charge/densité) et les **événements** actifs (ex. accident, urgence, fermeture), afin de privilégier un itinéraire adapté à l'état courant du réseau.

**Principe du coût :**
Le coût d'une route correspond à un **temps de trajet estimé** dérivé de ses attributs (longueur, vitesse maximale, capacité, trafic actuel et état). Lorsque la congestion augmente ou qu'un événement survient, la route devient plus pénalisante ; en cas de fermeture, elle est considérée **inutilisable** pour le calcul d'itinéraire.

**Mode Normal vs Mode Dynamique :**

En **mode normal**, les véhicules conservent leur chemin initial même si le réseau change. En **mode dynamique**, le système relance un calcul uniquement pour les véhicules réellement impactés, afin d'éviter les routes bloquées ou trop coûteuses.

**Recalcul en temps réel (mode dynamique) :**

1. Un itinéraire initial est calculé par le **PathPlanner** en utilisant **A*** (par défaut) ou **Dijkstra** (référence).
2. À chaque tick, la simulation met à jour la position des véhicules, le trafic par route et l'état des événements (durée restante, expiration).
3. Si une route du chemin devient indisponible ou fortement pénalisée, le véhicule est considéré comme *impacté*.
4. Seuls les véhicules impactés déclenchent une **replanification** via le **PathPlanner**, en tenant compte des nouveaux coûts.
5. Le véhicule remplace ensuite son itinéraire et poursuit sa progression sur le nouveau chemin.

**Répartition du trafic :**
Pour éviter qu'un reroutage massif ne crée un nouveau point de congestion, les coûts tiennent compte du trafic actuel, ce qui favorise une **diversification des chemins** et une meilleure fluidité globale.

**Justification A* / Dijkstra :**

- **Dijkstra** sert de référence : il calcule un chemin optimal en coût sans heuristique.
- **A*** est utilisé par défaut : il guide la recherche grâce à une heuristique (distance entre nœuds) afin d'accélérer le calcul.

**Mécanisme de reroutage dynamique**

Le **reroutage** est réalisé automatiquement **uniquement en mode dynamique**. Il est déclenché lorsqu'un **événement** affecte une route empruntée par un véhicule : le véhicule détecte alors que la route n'est **plus utilisable**, et la simulation lance un recalcul d'itinéraire pour les véhicules concernés. Concrètement, la méthode `rerouteAffectedVehicles(routeId)` parcourt la liste des véhicules ; si un véhicule se trouve sur la route impactée, la simulation récupère sa position courante (nœud actuel) ainsi que sa destination, puis demande au **PathPlanner** une **replanification** via `replanPath(currentPos, target, oldPath, currentRouteIndex)`. Le nouvel itinéraire est ensuite appliqué au véhicule avec `setPath(newPath)`, et un compteur de reroutages est mis à jour afin de produire des **statistiques** pendant l'exécution.

**Avantages :**

- **Adaptation en temps réel** : réaction automatique aux accidents et congestions.

- **Optimisation du temps** : sélection de trajets plus rapides (même si plus longs en distance).
- **Robustesse** : une route bloquée n'interrompt pas la simulation (chemin alternatif recalculé).
- **Comparaison claire** : mise en évidence des différences entre **mode Normal** et **mode Dynamique**.

### 3.2.2 Interface de simulation et visualisation (Raylib)

L'interface Raylib affiche la carte, les véhicules, et l'état des routes (normale / congestionnée / bloquée). Un panneau *Événements actifs* liste les perturbations en cours avec le temps restant, et une légende précise la signification des couleurs. Cela permet d'observer visuellement l'effet d'un accident sur les itinéraires et le reroutage automatique.


Fig. 3.1: Interface Raylib du sous-projet *Routage Dynamique* : visualisation des routes, véhicules, légende et événements actifs (ex. *ACCIDENT*) avec reroutage en temps réel.

## Représentation du réseau routier

Le réseau de la Smart City est représenté par un **graphe orienté** :

- Chaque **nœud (Node)** représente une intersection (avec ses coordonnées pour l'affichage).
- Chaque **route (Route)** représente une arête entre deux nœuds et possède des attributs utilisés par la simulation : longueur, vitesse maximale, capacité, état (active/bloquée) et **niveau de trafic**.

- Les perturbations sont modélisées sous forme **d'événements** (ACCIDENT, TRAFFIC_JAM, ROAD_CLOSURE, EMERGENCY) appliqués à une route avec une **sévérité** et une **durée**.

## Principe de Base

Le **routage dynamique** repose sur l'idée que le coût d'une route n'est pas constant : il évolue en fonction du trafic et des événements actifs. Nous cherchons donc l'itinéraire **le plus rapide** (temps estimé minimal) et non uniquement le plus court en distance. Le calcul de chemin est assuré par le module **PathPlanner** en utilisant une stratégie interchangeable (*Strategy*) : **A*** (par défaut) ou **Dijkstra** (référence).

## Étapes de Routage Dynamique

**A) Initialisation :**

1. Charger la configuration (JSON) : graphe, véhicules initiaux, paramètres de simulation.
2. Construire le **Graph** (nœuds + routes) et initialiser les structures de trafic.
3. Créer les véhicules et événements via des fabriques (**VehicleFactory** / **EventFactory**).
4. Choisir l'algorithme de pathfinding (A* ou Dijkstra) via la stratégie du **PathPlanner**.

**B) Calcul initial des itinéraires :**

1. Pour chaque véhicule, définir une source et une destination.
2. Calculer un premier trajet optimal (selon les poids courants du graphe).
3. Stocker le chemin (liste de nœuds) dans l'objet **Vehicle**.

**C) Mise à jour de la simulation (Update) :**

1. À chaque tick, mettre à jour la position des véhicules et estimer la densité/charge sur chaque route.
2. Mettre à jour les événements : décrémenter la durée restante et appliquer/retirer leurs effets.
3. Recalculer les **poids dynamiques** des routes (temps estimé = temps de base + pénalités trafic + pénalités événement).

**D) Détection d'impact :**

1. Vérifier si l'itinéraire courant d'un véhicule traverse une route devenue **bloquée** ou fortement pénalisée.
2. Marquer uniquement les véhicules concernés comme nécessitant un **reroutage**.

**E) Recalcul (Reroutage) :**

1. Relancer le **PathPlanner** avec les nouveaux poids (A* ou Dijkstra selon la stratégie active).
2. Obtenir un nouvel itinéraire minimisant le **temps de trajet** dans l'état actuel du réseau.
3. Mettre à jour le chemin du véhicule sans interrompre la simulation.

**F) Visualisation et suivi (Raylib) :**

- Afficher routes, véhicules et état des perturbations (panneau *Événements actifs* avec temps restant).
- Mettre en évidence les routes congestionnées/bloquées et observer l'adaptation des trajectoires.
- Permettre la comparaison **mode normal** vs **mode dynamique** durant la démonstration.

**Remarque :** Les classes sont séparées en fichiers d'en-tête (.h/.hpp) et d'implémentation (.cpp). La compilation est assurée par **CMake** (C++17) et la validation est renforcée par des **tests unitaires Catch2** sur les composants critiques (Graph, PathPlanner, Event, Vehicle, Simulation).

**Terminaison**

L'exécution du module de **routage dynamique** se termine lorsque la simulation est arrêtée (fin du scénario, arrêt manuel via l'interface) ou lorsque la durée totale configurée est atteinte. Durant l'exécution, chaque événement possède une **durée restante** : un événement est automatiquement retiré lorsqu'elle atteint zéro, puis les coûts des routes sont recalculés en conséquence. Ainsi, la simulation continue de manière stable jusqu'à l'arrêt, tout en garantissant la mise à jour continue des routes, des véhicules et des événements actifs.

## Gestion des paramètres de simulation

Notre simulateur de *Smart City* (Sous-projet 3 : *Routage dynamique*) a été conçu pour être **paramétrable** afin de tester différents scénarios de circulation et d'évaluer l'efficacité du reroutage en temps réel. Les paramètres sont définis dans des fichiers de configuration **JSON** (ex. `config/config.json`) et chargés au démarrage via *nlohmann/json*. Cette configuration permet notamment de définir la topologie du réseau (nœuds, routes et leurs propriétés), d'initialiser les véhicules, de préciser les paramètres globaux de la simulation, ainsi que les informations liées aux événements. L'intérêt de cette approche est de pouvoir modifier un scénario (réseau, flotte, événements) **sans modifier le code** et de garantir la reproductibilité des tests et des démonstrations. Pendant la démo Raylib, des contrôles permettent également d'interagir avec la simulation : `SPACE` déclenche un événement aléatoire, `R` bascule entre **mode Normal** et **mode Dynamique**, et `+ / -` ajuste le nombre de véhicules afin d'observer le comportement du système sous différentes charges.

## Gestion des Événements et Propriétés

La gestion des événements est centrale dans le routage dynamique, car elle modifie l'état du réseau et influence directement les itinéraires. Un événement (ex. *ACCIDENT*, *TRAFFIC_JAM*, *ROAD_CLOSURE*, *EMERGENCY*) est associé à une **route affectée**, une **sévérité** et une **durée**. Les événements sont créés via **EventFactory**, puis appliqués au réseau : selon le type, une route peut être pénalisée (ralentissement/augmentation du coût) ou bloquée. À chaque tick, la simulation met à jour la durée restante des événements ; lorsque celle-ci atteint zéro, l'événement expire, son effet est retiré, et les **poids dynamiques** des routes sont recalculés. Le reroutage est ensuite déclenché uniquement pour les véhicules dont l'itinéraire traverse une route devenue fortement pénalisée ou indisponible.

## Débogage et Visualisation

Pour valider la cohérence du comportement du routage dynamique, nous combinons **journalisation** et **visualisation**. Les logs permettent de suivre les événements actifs, l'état de certaines routes, ainsi que les recalculs d'itinéraires déclenchés pendant l'exécution. En parallèle, l'interface Raylib affiche le réseau et la circulation : routes, nœuds, véhicules, et un panneau *Événements actifs* indiquant le type d'événement et le **temps restant**. Cette visualisation rend observable l'impact d'un incident sur la circulation et la réaction du système en mode dynamique.

## Gestion des Véhicules

La classe **Vehicle** représente une entité mobile se déplaçant d'une intersection à une autre en suivant un itinéraire (liste de nœuds) calculé par le **PathPlanner**. À chaque mise à jour, un véhicule avance vers le prochain nœud de son chemin, met à jour son état (en mouvement, en attente, reroutage, arrivé) et vérifie si son itinéraire est impacté par un événement (route pénalisée ou bloquée). Si nécessaire, le véhicule signale à la simulation qu'un **recalcul d'itinéraire** est requis ; la simulation déclenche alors un reroutage via le planificateur, en utilisant les nouveaux poids du graphe.

## Classes principales impliquées dans le routage dynamique

- **Graph / Node / Route** : modélisation du réseau routier sous forme de graphe orienté (nœuds = intersections, routes = arêtes). Chargement depuis JSON et mise à jour des coûts dynamiques.
- **Event** : encapsule un événement (type, route affectée, sévérité, durée) et applique/retire son impact sur une route.
- **PathPlanner** : calcule l'itinéraire optimal en s'appuyant sur une stratégie interchangeable (*Strategy*) : **A*** ou **Dijkstra**.

- **Simulation** : orchestrateur central (tick), mise à jour des véhicules et événements, recalcul des poids, déclenchement du reroutage ciblé.
- **Renderer** : visualisation Raylib (réseau, véhicules, événements actifs, états des routes).
- **VehicleFactory / EventFactory** : création centralisée des véhicules et des événements (pattern *Factory*) pour garantir la cohérence et faciliter l'extension.

## Vue théorique générale

Le module de simulation du *routage dynamique* (notamment les classes **Vehicle**, **Simulation** et **Renderer**) repose sur plusieurs concepts clés :

**1. Gestion des états :**

Chaque véhicule peut se trouver dans différents états logiques (par exemple *en mouvement*, *en attente*, *en reroutage*, *arrivé*). Ces états déterminent son comportement à chaque tick : progression vers le prochain nœud, ralentissement si la route est congestionnée, ou déclenchement d'un recalcul d'itinéraire lorsque le chemin courant devient défavorable ou indisponible.

**2. Interaction utilisateur (mode démonstration) :**

L'utilisateur interagit avec la simulation via des contrôles dédiés afin d'observer l'impact du routage dynamique. Les commandes permettent notamment de déclencher des événements, de basculer entre **mode Normal** et **mode Dynamique**, d'ajuster la charge (nombre de véhicules) et de piloter l'exécution (pause/reprise) pendant la démonstration.

**3. Détection des contraintes de circulation :**

Le déplacement d'un véhicule est contraint par l'état du réseau. Une route peut être *normale*, *congestionnée* (coût élevé) ou *bloquée* (inaccessible). Avant d'emprunter une route, le véhicule (ou la simulation) vérifie sa disponibilité et son coût courant. Si une route du trajet devient bloquée ou trop pénalisante, le véhicule est identifié comme impacté et une procédure de reroutage est déclenchée.

**4. Rendu graphique (Raylib) :**

La visualisation s'appuie sur **Raylib** pour afficher le graphe routier (routes et intersections), les véhicules en mouvement et les informations d'état (légende, statistiques, événements actifs). Le rendu met en évidence l'évolution du trafic (routes normales / congestionnées / bloquées) afin de rendre immédiatement lisible l'effet des perturbations et des recalculs d'itinéraires.

**5. Gestion de la caméra (visualisation) :**

Pour améliorer la lisibilité, l'affichage peut utiliser une caméra 2D (centrage sur la carte ou suivi d'une zone d'intérêt) avec un éventuel zoom/déplacement. Cela facilite l'observation de la circulation et des changements de trajectoire lors du passage en mode dynamique, sans surcharger l'interface.

## Partie Technique :

**1. Attributs :**

- **Réseau et données** :
  - **graph** : instance du graphe orienté (nœuds/intersections et routes/arêtes).
  - **nodes / routes** : structures internes représentant la topologie et les propriétés des routes (longueur, vitesse, capacité, trafic, état utilisable/bloqué) Nœud central + 4 nœuds autour (N, S, E, W).
- **Entités de simulation** :
  - **vehicles** : flotte de véhicules (id, position actuelle, destination, chemin planifié, vitesse/type, détection du reroutage).
  - **events** : liste des événements actifs (type, route affectée, sévérité, durée/temps restant).
- **État global** :
  - **mode** : *Normal* (sans recalcul) ou *Dynamique* (avec reroutage).
  - **time / tick** : gestion du temps de simulation et des mises à jour (‘Route ::getTravelTime()‘).
  - **statistiques** : indicateurs (ex. nombre de reroutages, événements actifs, véhicules, etc.) pour l’analyse.
- **Affichage** :
  - **renderer** : module de rendu Raylib (routes, véhicules, événements, légende/infos).
  - **camera2D** : caméra 2D (déplacement + zoom) utilisée pendant la démonstration.

**2. Constructeur et Destructeur :**

- **Initialisation** : chargement du graphe, des véhicules initiaux et des paramètres via un parseur **JSON** (*nlohmann/json*) afin de permettre des scénarios modifiables sans recompilation.
- **Mise en place de la simulation** : allocation/initialisation des véhicules et événements (Factories), et préparation du rendu Raylib.
- **Gestion mémoire (RAII)** : utilisation de smart pointers (`std::unique_ptr` / `std::shared_ptr`) pour sécuriser la gestion des ressources dynamiques (véhicules, événements, stratégies).

**3. Déplacements et Contrôles (démo interactive) :**

- **SPACE** : déclencher un événement aléatoire pour perturber le réseau.
- **R** : basculer entre **mode Normal** et **mode Dynamique** afin de comparer les comportements.

- **+ / -** : augmenter / réduire le nombre de véhicules pour varier la charge de trafic.
- **Flèches / WASD** : déplacer la caméra.
- **Molette** : zoom avant / arrière.
- **ESC** : quitter l'application.

**4. Détection des contraintes de circulation :**

- **Routes utilisables** : vérification de l'état opérationnel d'une route (utilisable ou bloquée) avant de la considérer dans un chemin.
- **Poids dynamiques** : mise à jour des coûts en fonction du **trafic actuel** et des **événements** (sévérité/durée).
- **Détection du reroutage** : un véhicule identifie si son chemin planifié est impacté (coût excessif ou route bloquée) et signale la nécessité d'un recalcul.

**5. Rendu Graphique et Gestion de la Caméra :**

- **Rendu Raylib** : affichage du graphe (routes/intersections), des véhicules en mouvement, et des événements actifs avec un temps restant.
- **Lisibilité de l'état du trafic** : affichage distinctif des routes normales / congestionnées / bloquées et mise en évidence des adaptations en mode dynamique.
- **Caméra 2D** : déplacement et zoom pilotés par l'utilisateur pour observer la simulation sous différents points de vue.

## Gestion de l'état de la simulation

La classe **Simulation** centralise l'évolution temporelle du système : elle met à jour les événements (durée restante et expiration), recalcul les coûts dynamiques des routes, met à jour le déplacement des véhicules et déclenche le reroutage uniquement en **mode Dynamique**. Cette organisation permet une comparaison directe **Normal vs Dynamique** et met en évidence l'apport du recalcul d'itinéraires en présence de perturbations sur le réseau.

## Interface de simulation (Raylib)

L'interface graphique constitue un support essentiel pour observer et valider le comportement du **routage dynamique** en conditions perturbées. Elle est structurée autour d'une vue principale représentant le réseau routier (intersections et routes) et de panneaux d'information qui synthétisent l'état de la simulation en temps réel. Le panneau *Simulation* affiche les indicateurs globaux (temps, nombre de véhicules, événements actifs, nombre de reroutages,

mode courant et état d'exécution), tandis que le panneau *Événements actifs* liste les perturbations en cours avec leur **temps restant** (ex. *ACCIDENT*, *URGENCE*). Une légende précise la signification des états de route (normale, congestionnée, bloquée) afin de relier directement l'évolution visuelle du trafic au calcul des **poids dynamiques** et aux recalculs d'itinéraires.

## Gestion de l'exécution et coordination des modules

La classe **Simulation** joue le rôle de coordinateur central : elle orchestre la boucle principale, met à jour les entités (véhicules), applique l'évolution temporelle des événements (durée restante, expiration) et déclenche le reroutage lorsque le mode dynamique est activé. Le module **PathPlanner** fournit le calcul d'itinéraire (A* ou Dijkstra via une stratégie interchangeable) à partir des coûts courants du graphe, alors que **Graph** maintient la topologie et les propriétés des routes. Enfin, le **Renderer** assure la cohérence entre l'état interne de la simulation et l'affichage Raylib, en rendant visibles les véhicules, l'état des routes et les informations de diagnostic (statistiques, événements).

### En résumé

La conception et le développement du sous-projet ont suivi une approche **modulaire** et **orientée objet**, garantissant une séparation claire des responsabilités : **Graph** pour la modélisation du réseau, **Vehicle** pour le déplacement et le suivi d'itinéraire, **Event** pour les perturbations et leur impact, **PathPlanner** pour le calcul de chemin et **Simulation** pour l'orchestration globale. Cette organisation, couplée à la visualisation Raylib, permet de démontrer de manière claire l'intérêt du **mode dynamique** (adaptation des itinéraires en présence d'incidents) et de comparer le comportement du système selon différentes charges et perturbations, tout en conservant une base de code maintenable et évolutive.


Fig. 3.2: Interface Raylib du simulateur de routage dynamique : panneau de statistiques (simulation), liste des événements actifs et visualisation du réseau routier et des véhicules.

# Chapitre 4

# Fonctionnalités principales.

Pour implémenter les fonctionnalités principales du *simulateur de routage dynamique*, nous avons structuré le développement en plusieurs éléments clés. Chaque fonctionnalité est conçue pour permettre l'observation et l'évaluation du comportement des véhicules face aux perturbations du réseau, ainsi que la comparaison entre un fonctionnement sans reroutage et un fonctionnement adaptatif.

## 4.1 Simulation et calcul d'itinéraires

Notre sous-projet modélise la ville sous forme de **graphe orienté** : les **nœuds** représentent les intersections et les **routes** (arêtes) représentent les segments routiers. Le réseau est **chargé depuis un fichier JSON**, ce qui permet de modifier la topologie (nœuds, routes, propriétés) et les paramètres de simulation sans toucher au code source.

**Calcul d'un itinéraire initial :** À l'initialisation, les véhicules (source/destination) sont créés puis un **itinéraire initial** est calculé par la classe **PathPlanner**. Le planificateur s'appuie sur le **pattern Strategy** pour choisir l'algorithme de recherche : **A*** (par défaut, plus efficace) ou **Dijkstra** (référence pour la comparaison)

**Principe de coût (poids dynamique) :** Le routage ne cherche pas seulement un chemin "court", mais un chemin de **coût minimal** représentant un **temps estimé**. Ce coût intègre la congestion (trafic) et peut être modifié par des événements, ce qui rend certains chemins initialement optimaux **sous-optimaux** lorsque l'état du réseau évolue.

## 4.2 Gestion des modes (Normal / Dynamique)

La simulation permet de basculer entre deux modes afin de comparer le comportement du système lorsque le réseau change

- **Mode Normal (sans reroutage)** : les véhicules conservent le chemin calculé au départ. Même si un événement bloque ou pénalise une route, ils continuent vers cette route, ce qui peut provoquer des blocages, des files d'attente et un temps de trajet moyen élevé.

- **Mode Dynamique (avec reroutage)** : les véhicules détectent l'impact des événements sur leur itinéraire et déclenchent un **recalcul automatique** en tenant compte des conditions actuelles (trafic/événements), afin d'éviter les zones perturbées et d'améliorer la fluidité globale

  **Reroutage dynamique**

  En mode *Dynamique*, lorsqu'un événement affecte une route empruntée, la simulation identifie uniquement les véhicules concernés et déclenche une **replanification** via le **PathPlanner**. Le véhicule remplace alors son itinéraire par le nouveau chemin calculé à partir de sa position courante vers la destination, et le nombre de reroutages est mis à jour pour le suivi en temps réel.

- **Stratégie anti-blocage** : pour éviter qu'un reroutage massif ne crée un nouveau bouchon sur un même détour, le calcul prend en compte le **trafic en temps réel** et favorise une **diversification des chemins** (répartition des véhicules sur plusieurs alternatives).

**Mesure et comparaison :** pour comparer objectivement les deux modes, nous suivons notamment le **temps de trajet moyen**, le **nombre de reroutages**, et le **nombre de véhicules arrivés à destination**.

## 4.3 Gestion des événements de circulation

La simulation intègre un système d'événements appliqués à une route du graphe (création manuelle ou aléatoire). Un événement possède une **durée** et une **sévérité**, et son impact est visible pendant l'exécution.

- **Types d'événements** : *Accident*, *Embouteillage*, *Fermeture de route*, *Urgence*.
- **Application** : l'événement modifie l'état/la pénalité de la route affectée et reste actif jusqu'à l'expiration de sa durée.
- **Reroutage ciblé (mode dynamique)** : seuls les véhicules réellement impactés déclenchent un recalcul, ce qui maintient une simulation stable.

## 4.4 Contrôle de l'exécution et réglages

L'exécution est pilotée via des commandes simples (clavier), utiles pour déclencher des scénarios et observer le reroutage :

- **SPACE** : déclencher un événement aléatoire.
- **R** : basculer entre *Normal* et *Dynamique*.

- **+ / -** : augmenter / réduire le nombre de véhicules.
- **Flèches / WASD** : déplacer la caméra.
- **Molette** : zoom avant / arrière.
- **ESC** : quitter.

## 4.5 Interface graphique et feedback visuel (Raylib)

La visualisation Raylib fournit une lecture immédiate de l'état de la simulation (trafic, événements, reroutage) et une caméra interactive (déplacement/zoom).

- **Vue centrale** : rendu du réseau routier et des véhicules en mouvement.
- **Mise en évidence** : routes affectées (ex. routes bloquées visibles) et indicateur visuel des véhicules en reroutage
- **Statistiques en temps réel** : affichage du mode (NORMAL/DYNAMIC), nombre de reroutages et indicateurs permettant la comparaison.

# Chapitre 5

# Annexes

## 5.1 Diagramme de Classes

**Diagramme UML : *Graph*, *Route* et *Node***

La figure ci-dessous présente une vue simplifiée de la structure du réseau routier sous forme de graphe. La classe **Graph** centralise les collections de **nodes** (intersections) et de **routes** (liaisons entre intersections) et fournit les opérations principales d'ajout (`addNode()`, `addRoute()`) ainsi que la recherche d'itinéraire (`findPath()`). La classe **Route** décrit une liaison orientée entre deux nœuds (`fromNode` → `toNode`) et porte les propriétés nécessaires à la simulation (longueur, vitesse, capacité) ainsi que la méthode `isUsable()` permettant de vérifier si la route est praticable (ex. route bloquée par un événement). La classe **Node** représente une intersection identifiée par un `id` et localisée par ses coordonnées `x, y` utilisées notamment pour la visualisation Raylib.

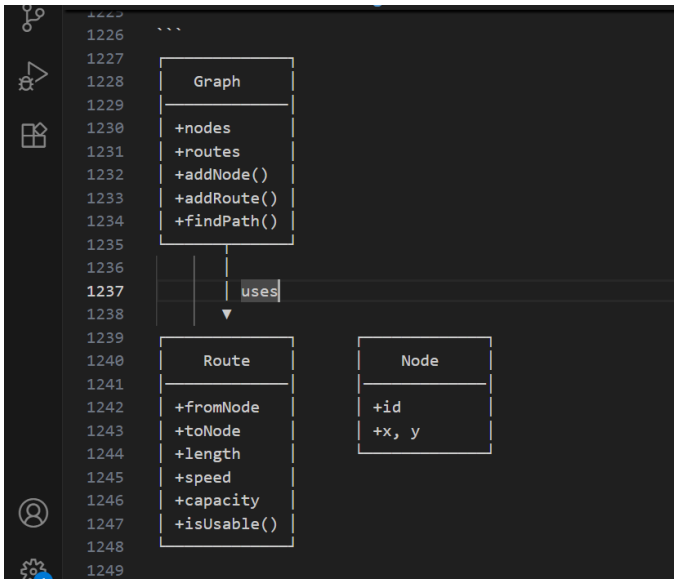


Fig. 5.1: Diagramme UML simplifié : relation entre *Graph*, *Route* et *Node*.

### Diagramme UML : *Simulation*, *Vehicle*, *Event* et *PathPlanner*

La figure suivante illustre l'architecture logique du sous-projet **Routage dynamique**. La classe **Simulation** joue le rôle d'orchestrateur : elle possède une référence vers le **graph** (réseau routier), maintient la liste des **vehicles** et des **events**, puis exécute la boucle de mise à jour via `update()`. Elle **gère** l'évolution des véhicules (déplacement, état, arrivée) ainsi que l'activation/expiration des événements (accident, urgence, etc.). La classe **Vehicle** stocke son identifiant `id`, l'itinéraire courant `path` et met à jour sa progression via `update()` ; en cas de perturbation, son trajet peut être remplacé via `setPath()`. La classe **Event** décrit une perturbation par son `type`, la route concernée `routeId`, sa `severity` et sa `duration`, ce qui impacte directement le coût ou l'accessibilité d'une route. Pour calculer ou recalculer un itinéraire, le véhicule (via la simulation) fait appel à **PathPlanner**, qui délègue la recherche à une stratégie **PathfindingStrategy** (méthode virtuelle pure `findPath()`). Deux implémentations sont utilisées : **AStarStrategy** (guidée par heuristique) et **DijkstraStrategy** (référence), permettant de comparer les comportements et performances.

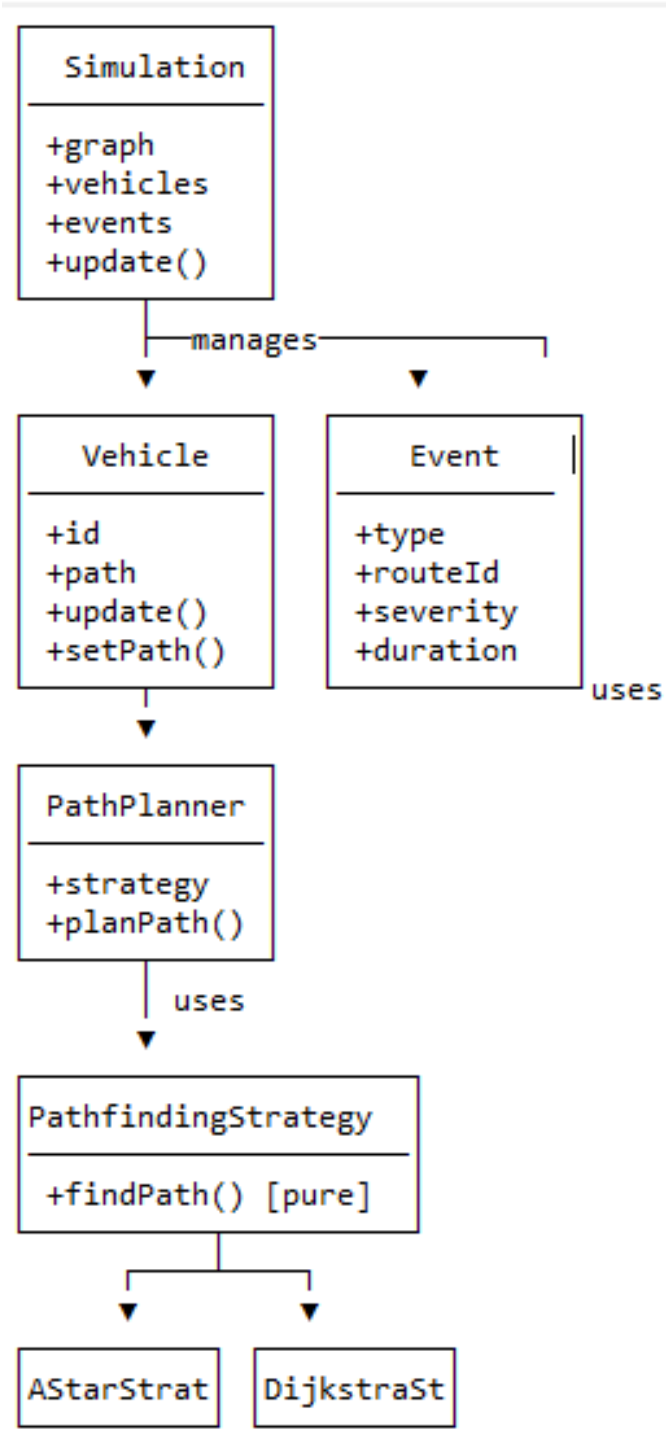


Fig. 5.2: Diagramme UML : orchestration de la simulation et planification d'itinéraires (Strategy : A* / Dijkstra).

### 5.1.1 Structure des classes

Le sous-projet *Routage Dynamique* est structuré de façon modulaire afin de séparer clairement la modélisation du réseau, la planification d'itinéraires, la simulation et le rendu graphique. L'arborescence principale est la suivante :

```
include/      # Déclarations (headers)
src/          # Implémentations (.cpp)
tests/        # Tests unitaires
demos/        # Application principale (main.cpp)
config/       # Paramètres JSON (config.json)
assets/       # Ressources (textures/sons)
external/     # Dépendances externes (raylib)
build/        # Fichiers compilés (généré)
```

Cette organisation facilite la maintenance du code et permet d'isoler les composants critiques (graphe, planification, événements, véhicules) des éléments liés à l'affichage.

### Relations entre classes (vue d'architecture)

L'architecture repose sur un orchestrateur central (**Simulation**) qui coordonne le graphe routier, la planification, les véhicules, les événements et le rendu :

- **Simulation** compose **Graph** et **PathPlanner**, et agrège des collections de **Vehicle** et **Event**.
- **Graph** compose des **Node** (intersections) et des **Route** (routes) pour représenter le réseau.
- **PathPlanner** s'appuie sur une **Strategy** (**PathfindingStrategy**) afin de choisir l'algorithme (**AStarStrategy** ou **DijkstraStrategy**).
- **Vehicle** consulte le graphe pour suivre un itinéraire et réagir aux modifications du réseau.
- **Event** affecte une **Route** (pénalité, ralentissement ou indisponibilité).
- **Renderer** utilise **Graph**, **Vehicle** et **Event** pour afficher l'état de la simulation avec **Raylib**.

# Chapitre 6

# Capture d'écran de la simulation

## 6.1 mode de simulation

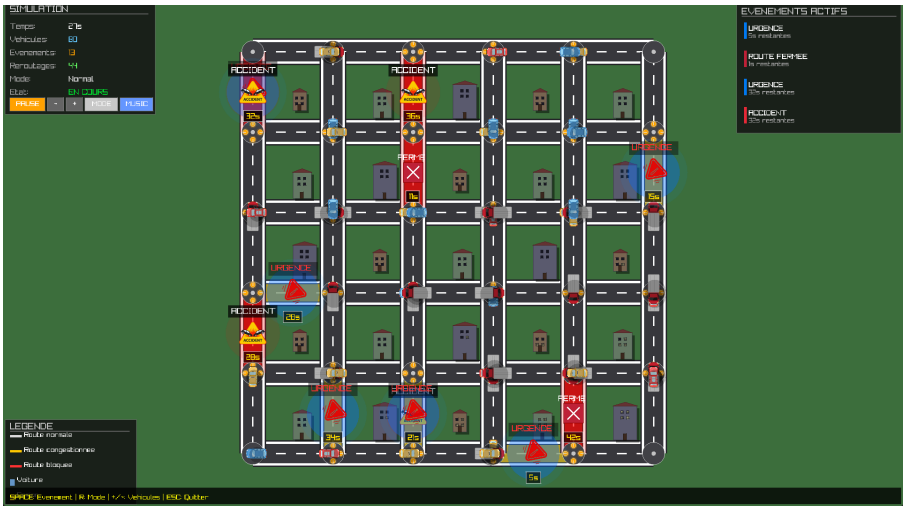


Fig. 6.1: Le mode normal

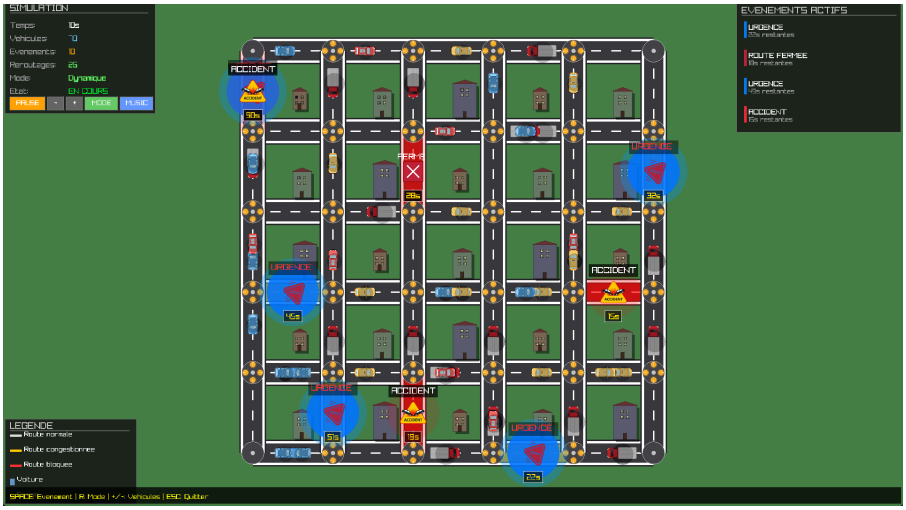


Fig. 6.2: Le mode dynamique

## 6.2 Détails de la simulation

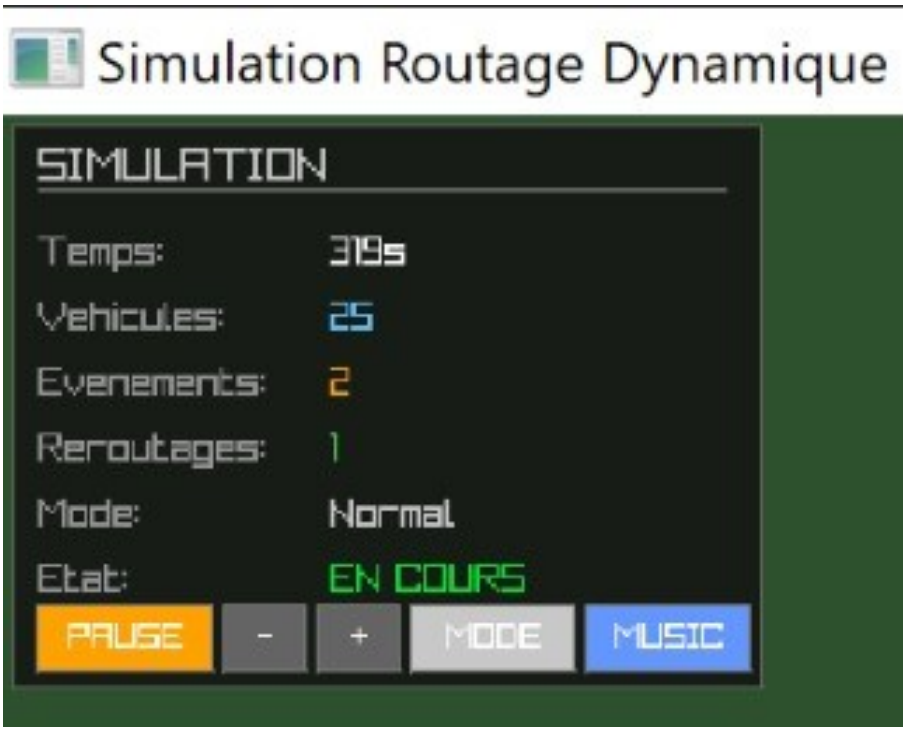


Fig. 6.3: Écran de détail de simulation


Fig. 6.4: Écran de détail de l'état courante de la route

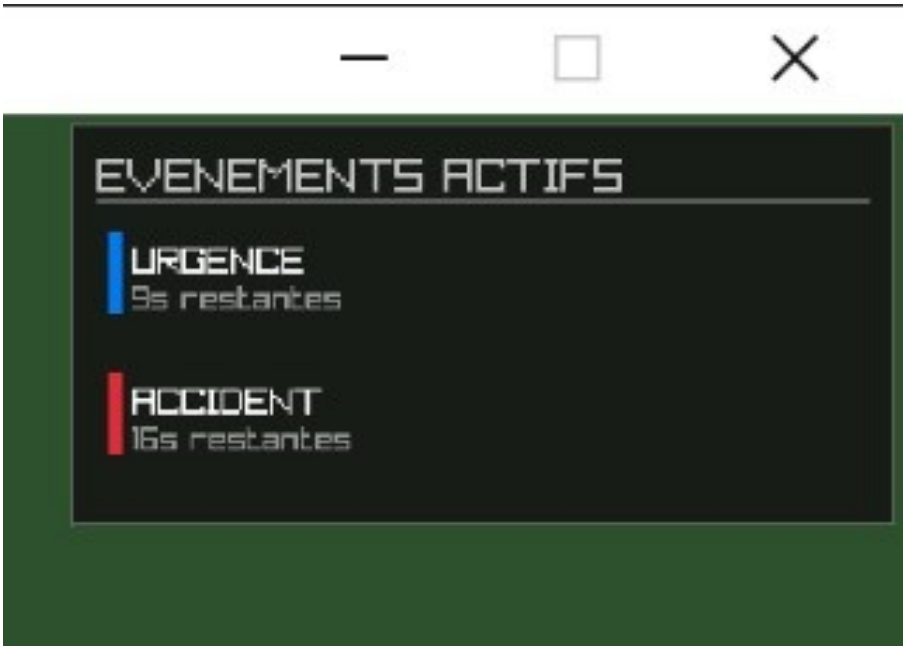


Fig. 6.5: Écran de détail des événements actifs


Fig. 6.6: Écran de confirmation de la simulation

# Chapitre 7

# Conclusion générale

## 7.1 Conclusion

Le sous-projet **Routage Dynamique** s'inscrit dans une démarche *Smart City* visant à améliorer la mobilité urbaine en adaptant les trajets en fonction de l'état réel du réseau. À travers une simulation interactive, nous avons mis en œuvre une approche complète combinant modélisation en **graphe routier**, gestion d'entités **Vehicle**, prise en compte d'événements (ex. *ACCIDENT*, *URGENCE*) et recalcul d'itinéraires en temps réel.

L'un des apports majeurs de ce travail réside dans l'intégration d'un **planificateur d'itinéraire** basé sur une stratégie interchangeable (*Strategy*), permettant d'utiliser **A*** ou **Dijkstra** selon le contexte. L'utilisation de **poids dynamiques** (coût évolutif des routes) rend la simulation plus réaliste : le système ne cherche pas uniquement le chemin le plus court, mais privilégie un itinéraire optimisé selon l'état courant (trafic et perturbations). La comparaison entre **mode Normal** et **mode Dynamique** met clairement en évidence l'intérêt du reroutage lorsqu'une route devient pénalisée ou indisponible.

Sur le plan logiciel, le projet illustre l'importance d'une conception **modulaire** et **orientée objet** favorisant la lisibilité, la maintenabilité et l'évolutivité. La séparation des responsabilités entre **Simulation** (orchestration), **Graph** (réseau), **PathPlanner** (calcul), **Event** (perturbations) et **Renderer** (visualisation Raylib) facilite l'extension du système. Le paramétrage via **JSON** permet en outre de varier rapidement les scénarios sans modification du code, et la visualisation Raylib offre un retour immédiat grâce aux panneaux de statistiques, à la liste des événements actifs et à la légende des états de routes.

En perspective, plusieurs améliorations peuvent renforcer ce sous-projet : l'ajout de scénarios plus complexes (réseaux plus larges), l'introduction d'une politique de reroutage plus fine (seuil de gain, limitation de changements successifs), ainsi que l'amélioration de l'analyse des résultats (statistiques détaillées, courbes d'évolution du trafic). Ces pistes prolongent naturellement l'objectif initial : disposer d'un simulateur clair, extensible et pertinent pour étudier l'adaptation dynamique des itinéraires dans un contexte Smart City.

# Bibliographie

**Raylib Website :** `https://www.raylib.com`

**Raylib Examples :** `https://www.raylib.com/examples.html`

**Raylib Cheatsheet :** `https://www.raylib.com/cheatsheet/cheatsheet.html`

**Raylib GitHub Repo :** `https://github.com/raysan5/raylib`

**Raylib GitHub Wiki :** `https://github.com/raysan5/raylib/wiki`

**C++ Reference :** `https://en.cppreference.com/w/`

**CMake Documentation :** `https://cmake.org/documentation/`

**nlohmann/json (JSON for Modern C++) :** `https://github.com/nlohmann/json`

**Catch2 (Tests unitaires) :** `https://github.com/catchorg/Catch2`