Rapport sur le projet :

Smart city using raylib

Sous projet : EMERGENCY & PRIORITY MANAGEMENT

Encadrée par :
P. Ikram Ben Abdel Ouahab
Réalisées par :
Amsaini Hiba
Farssi Fatima Zahra

TABLE DES MATIÈRES:

1. INTRODUCTION GÉNÉRALE

1.1 Contexte et Problématique

La gestion des flux urbains est devenue un enjeu majeur des villes modernes ("Smart Cities"). L'optimisation du trafic, la sécurité routière et, surtout, l'efficacité des services d'urgence (pompiers, police, ambulances) nécessitent des modèles de simulation performants.

Dans ce contexte, notre projet consiste à développer un simulateur de ville en temps réel. La problématique centrale est la suivante : Comment modéliser une interaction fluide entre des agents autonomes respectant le code de la route et des agents prioritaires devant enfreindre ces règles pour répondre à des situations de crise ?

1.2 Cahier des charges

Le simulateur doit répondre aux exigences suivantes :

- Graphisme : Vue 2D "Top-Down" (vue de dessus) claire et lisible.
- Dynamisme : La ville doit vivre (circulation continue, feux tricolores fonctionnels).
- Interactivité : L'utilisateur doit pouvoir déclencher des urgences ou modifier l'environnement (Cycle jour/nuit).
- Intelligence : Les véhicules ne doivent pas entrer en collision et doivent avoir des comportements distincts selon leur type.

1.3 Objectifs Pédagogiques

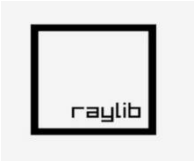
Ce projet vise à consolider nos compétences en :

- Programmation Orientée Objet (Classes, Héritage, Polymorphisme).
- Gestion de la mémoire en C++ (Pointeurs, vecteurs dynamiques std::vector).

- Mathématiques appliquées (Vecteurs 2D, calculs de distance, interpolation).

2. ENVIRONNEMENT DE DÉVELOPPEMENT :

2.1Les outils utilise

	est un éditeur de code extensibledéveloppé par Microsoft pour Windows, Linux et macOS.
	Raylib est une bibliothèque graphique open-source en C, conçue pour créer des jeux 2D et 3D facilement. Elle est simple à utiliser, légère et idéale pour les débutants, tout en offrant des fonctionnalités puissantes pour les développeurs expérimentés.
	est un outil multiplateforme qui génère des fichiers de build (Makefiles, Visual Studio, etc.) à partir de fichiers CMakeLists.txt, simplifiant la configuration et la compilation des projets C/C++.
	est une plateforme de gestion de code basée sur Git, permettant de collaborer, versionner et héberger des projets. Elle est idéale pour les développeurs et équipes pour partager, réviser et déployer du code.

2.2 Le Langage C++

Nous avons choisi le C++ pour sa robustesse et sa rapidité d'exécution, critères essentiels pour une simulation en temps réel affichant de nombreuses entités. L'utilisation de la STL (Standard Template Library), notamment pour les vecteurs, a grandement facilité la gestion dynamique des listes de voitures.

2.3 La Bibliothèque Raylib

Pour l'affichage graphique, nous avons opté pour Raylib. Contrairement à des moteurs lourds comme Unity ou Unreal, Raylib est une bibliothèque légère orientée code. Elle permet de gérer :

- Le dessin de formes primitives (Rectangles, Cercles, Lignes).
- La gestion des entrées (Clavier, Souris).
- Le calcul vectoriel via raymath.h.
- Les collisions de base (CheckCollisionRecs).

3.ANALYSE ET CONCEPTION TECHNIQUE

3.1 Architecture Modulaire

Au lieu de tout coder dans un seul fichier géant, ce qui aurait rendu le débogage impossible, nous avons séparé le projet en deux dossiers principaux : src (pour le code source) et include (pour les fichiers d'en-tête).

- **Le chef d'orchestre (main.cpp) :** C'est le point d'entrée du programme. Son rôle est unique : il initialise la fenêtre graphique, lance la boucle de jeu infinie et capture les actions du joueur (clavier et souris). Il ne contient pas de logique complexe, il se contente d'appeler les fonctions des autres fichiers.
- **L'architecte (world.cpp) :** Ce fichier est responsable de la construction de la ville. C'est lui qui calcule mathématiquement où placer les routes et les bâtiments en fonction de la taille de l'écran.

- **L'intelligence (vehicle.cpp) :** C'est le module le plus dense. Il contient toute la logique de déplacement des voitures, la détection des collisions et la gestion des missions d'urgence.
- **La gestion du temps (traffic_system.cpp) :** Ce module gère tout ce qui est temporel : le changement de couleur des feux tricolores toutes les 3 secondes et l'apparition aléatoire des accidents.
- **Module Engine :** Gère le rendu UI et les utilitaires.

3.3 Pourquoi on a fait ces choix ?

1. Pour gérer la liste des voitures (std::vector) Au lieu d'utiliser un tableau fixe (qui a une taille limitée), on a utilisé des **vecteurs**. C'est une liste intelligente qui peut s'agrandir ou rétrécir.

- *Avantage :* Si on veut ajouter 10 voitures d'un coup ou en supprimer une qui rentre au garage, le jeu ne plante pas.

2. Pour régler le jeu facilement (config.h) On a mis tous les réglages du jeu dans un petit fichier à part appelé config.h.

- *Exemple :* Si on trouve que les voitures roulent trop lentement, on change juste le chiffre CAR_SPEED dans ce fichier, et ça change la vitesse de toutes les voitures d'un coup. Pas besoin de chercher dans tout le code.

3.2 Modélisation des Classes

Le cœur du projet repose sur la structure **Car.**

Attributs principaux de la classe Car : Au lieu de gérer des variables séparées, chaque voiture est un objet complet. Elle connaît : Sa position (où elle est) ,Sa vitesse (à quelle allure elle roule), Son rôle (si c'est une voiture de police ou un civil), Son état (si elle est en pause ou en mission).

- **Vector2 pos** : Position cartésienne (x, y).
- **Dir dir** : Enumération (UP, DOWN, LEFT, RIGHT).

- **Type type** : Définit le comportement (CIVIL vs URGENCES).
- **EmergencyState emState** : Variable d'état pour la gestion des missions.

Structure Building : Contrairement aux voitures, les bâtiments sont statiques mais possèdent des métadonnées essentielles : un entryPoint (point de sortie vers la route) et un center (point visé par les véhicules de secours).

3.3 Modélisation Mathématique de la Ville

La ville n'est pas une image fixe, mais une grille calculée dynamiquement.

- Les routes verticales sont stockées dans `std::vector<float> vRoads`.
- Les routes horizontales sont stockées dans `std::vector<float> hRoads`.

- Une intersection est définie par le point de rencontre `(vRoads[i], hRoads[j])`.

Cette approche mathématique permet d'utiliser la fonction `GetSnapAxis()` qui "aimante" les voitures sur l'axe le plus proche, garantissant qu'elles restent parfaitement au centre de leur voie.

4. IMPLÉMENTATION DES ALGORITHMES

4.1 Génération Procédurale (RecalculateGrid)

Au lancement, le programme analyse la résolution de l'écran

(ex: 1300x700). Il divise ensuite cet espace par TARGET_BLOCK_SIZE (220px) pour déterminer combien de routes peuvent être créées.

Cela rend le simulateur "Responsive" : si on agrandit la fenêtre, la ville s'agrandit, de nouvelles routes apparaissent et les bâtiments se replacent automatiquement aux coins stratégiques.

4.2 Intelligence Artificielle et Navigation

L'IA repose sur un système **de détection d'obstacles par anticipation** (Raycasting simplifié).

Le concept du "Sensor" : Chaque voiture projette un rectangle invisible devant elle. La taille de ce rectangle dépend de la vitesse du véhicule (plus il va vite, plus il regarde loin).

```
// Formule simplifiée du capteur
float lookAhead = 70.0f + (speed * 25.0f);
```

Si ce rectangle intersecte (CheckCollisionRecs) le rectangle physique d'une autre voiture, le véhicule freine. Cela permet de créer naturellement des files d'attente aux feux rouges sans coder explicitement une "file".

4.3 Gestion des Feux Tricolores

Les feux fonctionnent sur un cycle global partagé par toutes les entités :

1. V_GREEN : Voies verticales passent.
2. V_YELLOW : Voies verticales ralentissent.
3. H_GREEN : Voies horizontales passent.
4. H_YELLOW : Voies horizontales ralentissent.

Une voiture civile vérifie deux conditions pour s'arrêter :

***Est-ce que je fais face au feu ? (Direction vs Axe).**

***Est-ce que le feu est rouge ?**

4.4 Machine à États des Services d'Urgence

C'est l'algorithme le plus complexe du projet. Un véhicule de pompier ou d'ambulance suit le diagramme d'états suivant :

1. **IDLE (Attente)** : Le véhicule est invisible ou garé.
2. **ALERTE (Trigger)** : Un événement (feu/accident) est détecté.
3. **DEPLOYING (Déploiement)** : Le véhicule sort du bâtiment vers la route (entryPoint).
4. **ON_MISSION (En route)** : Le véhicule navigue vers les coordonnées de l'incident. Il ignore les feux rouges et roule plus vite (maxSpeed = 4.0f).
5. **ACTION (Extinguishing/Treating)** : Une fois à moins de 70px de la cible, il s'arrête et simule une intervention (particules, délais).
6. **RETURNING (Retour)** : La mission finie, il calcule le chemin vers son homeEntry.
7. **DOCKING (Stationnement)** : Il rentre dans le bâtiment et disparaît.

5. INTERFACE ET EXPÉRIENCE UTILISATEUR

5.1 Cycle Jour / Nuit

Pour augmenter le réalisme, nous avons implémenté un système de rendu conditionnel.

- En mode **Jour** : Fond vert (`COLOR_GRASS`), éclairage global neutre.
- En mode **Nuit** : Ajout d'un rectangle noir semi-transparent (`Alpha = 0.7`) sur toute la carte pour simuler l'obscurité.
- **Gestion des lumières :** Pour simuler les phares, nous utilisons le "**Mode Additif**" ou des dégradés de transparence (`Fade`) pour dessiner des triangles jaunes/bleus devant les voitures, qui sont dessinés par-dessus le filtre nuit.

5.2 Mini-Map (Radar)

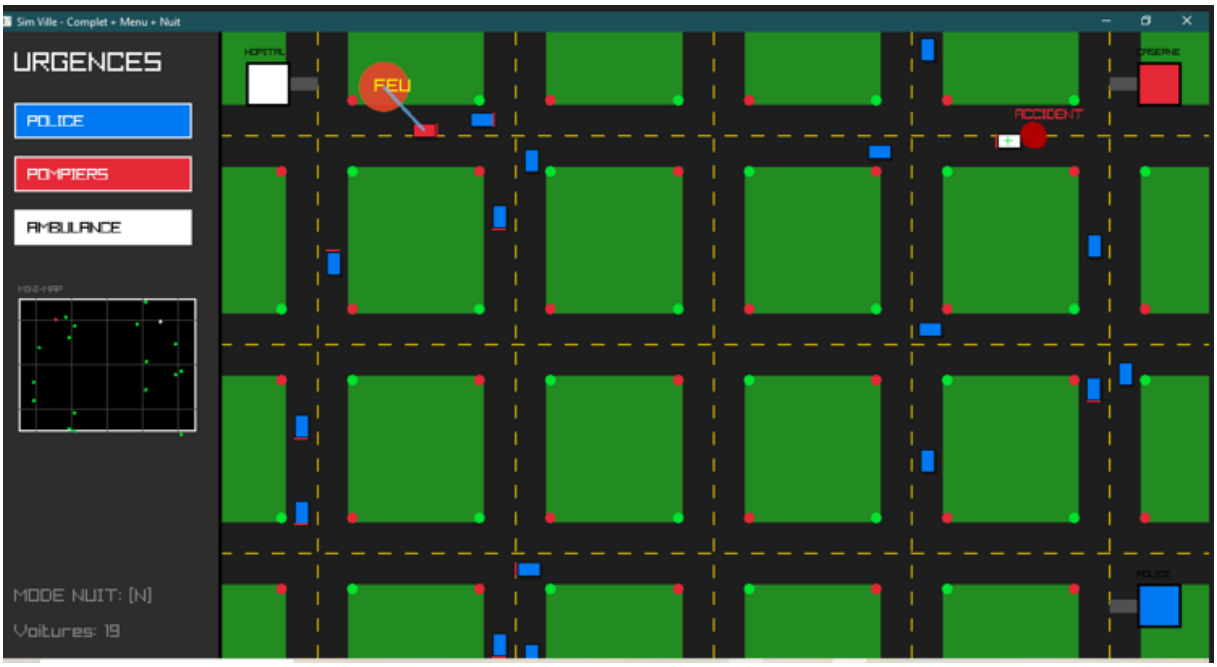
Afin de donner une vue d'ensemble, une mini-carte est dessinée en bas à gauche. Elle utilise une transformation mathématique simple (règle de trois) pour convertir les coordonnées du monde (0 à 1300) vers les coordonnées de la mini-map (0 à 200).

6.RÉSULTATS ET TESTS

Le projet a été testé sous différentes conditions :

- **Test de charge** : Jusqu'à 50 véhicules simultanés sans baisse notable de FPS (Frames Per Second).
- **Test de collision** : Les voitures ne se chevauchent pas, même en cas de freinage d'urgence.
- **Test de priorité** : Lorsqu'une ambulance arrive, les voitures civiles effectuent bien l'action `isYielding` (se décaler sur le trottoir) pour laisser le champ libre.

**Comme l'illustre la Figure , l'architecture du projet s'articule autour de la classe centrale Car. On remarque que cette classe n'agit pas seule : elle interagit avec la structure Building pour déterminer ses points de départ et d'arrivée, ainsi qu'avec le module TrafficLight pour respecter la signalisation. Ce découpage permet d'isoler la logique de mouvement de la gestion de l'environnement."



7. CONCLUSION ET PERSPECTIVES

7.1 Bilan

Ce projet de "Smart City Simulator" a été un défi technique enrichissant. Il nous a permis de comprendre la complexité cachée derrière des systèmes qui semblent simples en apparence, comme la gestion d'un carrefour. L'utilisation du C++ nous a forcés à être rigoureux sur la structure des données.

7.2 Améliorations futures

Le simulateur est fonctionnel, mais plusieurs axes d'amélioration sont envisageables :

- ***Pathfinding avancé (A)** :** Actuellement, les voitures tournent parfois de manière aléatoire ou simpliste. Un algorithme A* permettrait de trouver le chemin le plus court réel.
- **Piétons** : Ajouter des agents piétons traversant sur les passages cloutés.
- **Statistiques :** Enregistrer le temps moyen d'intervention des secours pour optimiser l'emplacement des casernes.

Ce projet constitue une base solide pour un système de simulation urbaine plus vaste.

8. BIBLIOGRAPHIE

- Documentation officielle de Raylib (raylib.com)
- Cours de C++ sur OpenClassrooms / LearnCpp
- Documentation sur les automates finis (State Machines)