LST IDAI 2025-2026

كلية العلوم و التقنيات بطنجة

ⵜⴰⵖⵡⵍⵓⵜ ⵏ ⵜⵎⴰⵙⵙⴰⵏⵜ ⴰ ⵜⵓⵏⵖⵣⵜⵏ ⵏ ⵟⴰⵏⵊⴰ

Faculté des Sciences et Techniques de Tanger

TRAFFIC CORE :

CONCEPTION D'UNE PLATEFORME DE SIMULATION 3D, NAVIGATION PAR GRAPHES, PATHFINDING ET RÉGULATION DE TRAFIC

Présenté par :

KHBIAZ ZINEB — **Groupe 3**

CHOROUK MOUBSSIR — **Groupe 3**

OUSSAMA JABIR — **Groupe 3**

AYA EL ISSAOUI — **Groupe 3**

Encadré par :

Pr. Ikram BENABDELOUAHAB

1. VUE D'ENSEMBLE DU PROJET

Ce projet est une simulation en temps réel d'un environnement urbain ("Smart City"). Le système gère un réseau routier complexe, des bâtiments 3D et une flotte de véhicules autonomes capables de naviguer d'un point A à un point B en calculant le chemin optimal et en évitant les collisions dynamiques.

L'architecture a été conçue de manière **modulaire**, séparant la logique de données, la logique de simulation et l'interface utilisateur.

2. ARCHITECTURE LOGICIELLE ET DESCRIPTION DES CLASSES

Le projet est structuré autour de 6 fichiers principaux. Voici le détail technique de chaque classe et structure.

A. Node.h (Structure de Données)

Cette structure représente l'unité atomique du graphe de navigation.

- **Rôle :** Représenter un point géographique dans la ville (intersection, virage, arrêt).
- **Attributs :**
    - std::string id : Identifiant unique (ex: "A", "G1").
    - Vector3 position : Coordonnées spatiales (x, y, z).
    - NodeType type : Enumération (DIRECT, INVERSE, CIRCULAR) pour la couleur de débogage.
    - std::vector<string> neighbors : Liste d'adjacence contenant les ID des nœuds accessibles depuis celui-ci.

B. CityGraph.h (Gestion du Réseau Routier)

Cette classe encapsule la topologie de la ville.

- **Attributs :**
    - std::map<string, Node> nodes : Conteneur associatif permettant un accès en O(1) (temps constant) aux nœuds via leur ID.
- **Méthodes Clés :**
    - AddNode() : Instancie un nœud et l'ajoute à la map.
    - Connect(A, B) : Crée une arête dirigée (route à sens unique) de A vers B.
    - GetShortestPath() : Implémentation de l'algorithme de **Dijkstra**. Retourne un vector<Vector3> représentant la suite de positions à suivre.

- GetNodeClicked() : Utilise le **Raycasting** (lancer de rayon) pour détecter si la souris survole un nœud 3D.

C. Vehicles.h (Polymorphisme et Agents)

Ce fichier définit les agents mobiles via l'Héritage.

Classe Mère : Vehicle

- **Attributs :**
  - position (Vector3), rotation (float), speed (float).
  - path : La liste des points à parcourir.
  - currentPathIndex : L'étape actuelle du trajet.
- **Méthodes :**
  - Update(bool isPaused) : Calcule le vecteur directionnel vers le prochain point, normalise ce vecteur, et applique la vitesse. Met à jour la rotation via atan2 pour orienter le véhicule.
  - Draw() : Affiche le modèle 3D avec la bonne échelle et rotation.

Classes Filles : Taxi, Van, Truck

- **Spécificité :** Elles héritent de Vehicle mais surchargent le constructeur pour définir des caractéristiques uniques :
  - **Taxi :** Rapide (0.075f), petite taille (0.5f).
  - **Van :** Vitesse moyenne, taille moyenne.
  - **Truck :** Lent (0.04f), grande taille (0.8f).

D. Simulation.h (Moteur Physique et Logique)

C'est le contrôleur principal (le "Cerveau").

- **Attributs :**
  - vehicles : Vecteur dynamique de pointeurs Vehicle* (permettant le polymorphisme).
  - orders : File d'attente des commandes de spawn.
- **Méthodes Clés :**
  - InitGraph() : Contient toute la configuration "Hardcodée" de la carte (positions des routes et connexions).
  - **Update() (CRITIQUE)** : Gère la boucle de jeu.

- Déplace chaque véhicule.
- **Système Anti-Ghosting :** Vérifie pour chaque paire de véhicules si :
    1. La distance < **4.0f**.
    2. Le véhicule B est dans le cône de vision du véhicule A (Produit Scalaire > 0.5).
    3. Les deux véhicules vont dans la même direction (Différence d'angle faible).
    - *Action :* Si vrai, le véhicule arrête de bouger.

E. CityUI.h (Interface Utilisateur)

Gère l'interaction 2D (HUD et Menus).

- **États :** Gère la machine à états (MENU_CONFIG vs SIMULATION_RUNNING).
- **Attributs :** Rectangles de collision pour les boutons et zones de texte.
- **Méthodes :**
    - HandleInput() : Gère la saisie clavier (remplissage des champs texte char par char) et les clics souris.
    - DrawMenu() : Affiche l'interface de configuration initiale.
    - DrawHUD() : Affiche les infos en temps réel (nombre de véhicules, FPS).

F. main.cpp (Point d'Entrée)

- **Rôle :**
    - Initialise la fenêtre Raylib et OpenGL.
    - Charge les Assets (Modèles 3D .glb) en mémoire GPU.
    - Contient la **Game Loop** (Boucle infinie while !WindowShouldClose).
    - Orchestre les appels à ui.HandleInput, sim.Update, et BeginMode3D pour le rendu.

G. TrafficManager.h / .cpp (Gestion de la Signalisation)

Cette classe gère l'infrastructure statique et temporelle des intersections.

⇨ **Rôle :** Réguler le flux de trafic pour éviter les blocages aux intersections majeures

⇨ **Attributs :**

- **lights (std::vector<TrafficLightInstance>) :** Conteneur stockant chaque feu tricolore avec sa position et son orientation.
- **timer (float) :** Compteur de temps global pour le cycle de changement.
- **switchTime (float) :** Durée d'un cycle complet (ex: 4.0f secondes) avant de passer à l'état suivant.

⇨ **Méthodes Clés :**

- **Update(float deltaTime) :** Gère la machine à états finis des feux. Elle assure la transition cyclique : `VERT -> JAUNE -> ROUGE -> VERT`.
- **Draw(Model& model) :** Effectue le rendu des modèles 3D des feux. La couleur du modèle (Teinte/Tint) change dynamiquement en fonction de l'état (`LIME` pour vert, `GOLD` pour jaune, `RED` pour rouge).

H. Système de Gestion de Caméras (Multi-vues)

=> Ce module permet à l'utilisateur de changer de perspective en temps réel pour analyser le trafic sous différents angles.

- **Rôle :** Offrir une flexibilité d'observation (vue globale vs immersion).
- **Les Modes Intégrés :**
  - **Vue Libre (Touche 6) :** Utilise UpdateCamera en mode CAMERA_FREE. L'utilisateur navigue avec Z, Q, S, D et la souris.
  - **Vue Carte / Zénithale (Touche 5) :** Positionne la caméra à une altitude fixe ($y=35$) avec une cible au centre ($0,0,0$). Le vecteur up est ajusté à $\{0, 0, -1\}$ pour une vue 2D parfaite.
  - **Vue Drone / Suivi (Touche S) :** La caméra se verrouille sur un véhicule (followedVehicle)

3. ANALYSE DES ALGORITHMES COMPLEXES

1. Pathfinding (Dijkstra)

L'algorithme utilisé dans CityGraph::GetShortestPath garantit le chemin le plus court.

- **Structure de données :** std::priority_queue (File de priorité).
- **Fonctionnement :**
  1. Initialiser toutes les distances à l'infini, sauf le départ (0).
  2. Tant que la file n'est pas vide, extraire le nœud avec la plus petite distance.
  3. Explorer ses voisins. Si Distance(Depart->Voisin) est plus courte que celle connue, mettre à jour la distance et noter le "Parent".

4. À la fin, remonter la chaîne des "Parents" depuis l'Arrivée vers le Départ, puis inverser la liste (std::reverse).

2. Mathématiques Vectorielles (Mouvement)

Le déplacement n'est pas "téléporté" mais calculé vectoriellement pour être fluide :

$$\vec{Direction} = \text{Normalize}(\vec{Cible} - \vec{Position})$$

$$\vec{NouvellePosition} = \vec{Position} + (\vec{Direction} \times \text{Vitesse})$$

$$\text{Angle} = \text{atan2}(\vec{Direction}.x, \vec{Direction}.z)$$

3. Logique de Signalisation et Synchronisation Le `TrafficManager` utilise une synchronisation temporelle simple mais efficace. Chaque instance de feu possède son propre décalage initial ou état de départ, permettant de créer des "ondes vertes" ou des alternances entre deux axes routiers.

- **Algorithme de cycle :**
  1. Si `timer > switchTime` : Réinitialiser `timer`.
  2. Pour chaque feu : `État_Actuel = (État_Actuel + 1) % 3`.

4. Calcul Trigonométrique de la Caméra Drone Pour que la caméra suive le véhicule tout en restant derrière lui, nous utilisons les fonctions `sinf()` et `cosf()` basées sur l'angle de rotation du véhicule. Cela permet d'obtenir un vecteur de recul qui tourne en même temps que la voiture, garantissant que l'utilisateur voit toujours l'avant du trajet.

4. GUIDE D'UTILISATION

Phase 1 : Configuration (Menu)

1. **Limites :** Définir le nombre max de véhicules (ex: 50).
2. **Sélection :** Choisir le type (Taxi, Van ou Camion).
3. **Trajet :** Entrer l'ID de départ (ex: "A") et d'arrivée (ex: "I").
4. **Quantité :** Nombre de véhicules à générer pour ce trajet.
5. Cliquez sur **INSÉRER** pour valider l'ordre, puis **DÉMARRER**.

Phase 2 : Simulation

- **Caméra Libre :** Z, Q, S, D + Souris.
- **Caméra Drone :** Touche **S** (ou clic) sur un véhicule pour le suivre automatiquement.

Touche	Action	Usage
5	**Vue Carte**	Observer l'ensemble du réseau routier et des nœuds.
6	**Vue Libre**	Se déplacer manuellement dans la ville 3D.
S	**Vue Drone**	Suivre automatiquement le véhicule sélectionné.
Espace	**Pause**	Geler la simulation pour analyser une situation.

- **Interaction :**
  - **Clic Gauche** sur un nœud : Ajoute un véhicule manuellement.
  - **Clic Droit** sur un véhicule : Le supprime de la simulation.

Observation du Trafic : Les feux tricolores fonctionnent de manière autonome dès le démarrage.

- **Visuel :** La base ou la lanterne du feu change de couleur pour indiquer l'autorisation de passage.
- **Échelle :** Les modèles sont rendus à 50% (`0.5f`) de leur taille originale pour une intégration visuelle parfaite avec les bâtiments.

5. CONCLUSION

=> Ce projet démontre une application concrète de la programmation C++ avancée :

- Utilisation extensive de la **STL** (vector, map, string, queue).
- Architecture **Orientée Objet** propre (Encapsulation, Héritage, Polymorphisme).
- Gestion de problèmes mathématiques et logiques en temps réel (collisions, navigation).

⇨ Le code est désormais modulaire, robuste et intègre déjà des fonctionnalités avancées comme la **gestion de la signalisation dynamique (Traffic Lights)**, posant les bases d'une simulation de conduite autonome encore plus réaliste (gestion des arrêts aux feux, détection de files d'attente, etc.).

6. DIAGRAMME DE CLASS :