# Smart city
# Avec raylib et c++

# Rapport de projet


***Réalisée par :***

***IMANE RHANEBOU***

***MOHAMED ZARKIK***

***MOHAMED ALAOUI***

***ENCADRÉPAR : M.IKRAM BENABDEL OUAHAB***

# SOMMAIRE :

# 1.introduction générale :

Le concept de **Smart City** (ville intelligente) désigne un environnement urbain moderne qui s'appuie sur l'intégration stratégique de la **technologie** et du traitement de données pour optimiser la **mobilité**, la **gestion des ressources** énergétiques et infrastructurelles, ainsi que la **qualité de vie** globale des citoyens. Plutôt que de traiter les services urbains de manière isolée, la Smart City les transforme en un écosystème interconnecté et réactif aux besoins en temps réel.

Dans le cadre de ce projet, l'objectif principal est de concevoir un **simulateur de Smart City modulaire** et évolutif. Cette plateforme a pour mission de reproduire fidèlement la complexité de la **mobilité urbaine** à travers plusieurs piliers technologiques :

- **Régulation du trafic :** Modéliser une circulation routière réaliste sur un réseau de routes et d'intersections configurables.
- **Intelligence infrastructurelle :** Mettre en œuvre une gestion automatisée des feux de signalisation capable de s'adapter aux flux de véhicules.
- **Adaptabilité dynamique :** Intégrer des algorithmes de reroutage pour contourner les obstacles ou les embouteillages imprévus.
- **Logistique de stationnement :** Développer une gestion intelligente des **parkings** permettant d'optimiser l'occupation de l'espace urbain.
- **Priorisation des services :** Assurer le traitement spécifique des véhicules prioritaires (secours, police) pour garantir leur efficacité en situation d'urgence.

En combinant ces différents modules, le simulateur permet d'étudier comment l'interaction entre les infrastructures (comme les parkings) et les usagers peut fluidifier durablement la vie urbaine.

# Les outils utilisé :

GitHub 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.


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.


CMake 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++.

Visual Studio Code est un éditeur de code extensible développé par Microsoft pour Windows, Linux et macOS.

# Objectifs

Le projet s'articule autour de l'idée de concevoir un système de gestion intelligente du stationnement au sein d'une Smart City modulaire. Le simulateur doit permettre de gérer la capacité des parkings en temps réel, d'orienter les véhicules vers des places libres selon diverses stratégies (proximité, coût) et de réguler les flux d'entrées et de sorties. L'utilisation de la bibliothèque Raylib pour la visualisation graphique, couplée à la gestion de scénarios complexes comme la saturation des infrastructures, enrichit l'interaction et la compréhension de la mobilité urbaine.

## Langage de programmation

Le jeu est développé en C++ pour ses performances et sa flexibilité

## Paradigm utilisé

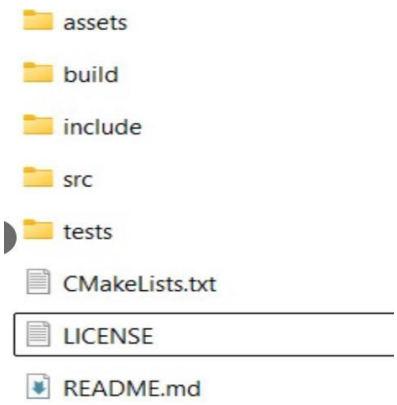
Dans ce jeu on a utilisé la Programmation Orienté Objet pour assuré la réutilisation du code, l'évolutivité et l'efficacité.

## Textures et effets

Des textures spécifiques ont été conçues pour les routes et les voitures , avec une perspective 2D

# 2. conceptions et architecture :

***STRUCTURE DU PROJET :***




***Assets :***Ce répertoire contient toutes les ressources non-logicielles nécessaires à la simulation. On y retrouve les images (textures pour les parkings et véhicules), les polices d'écriture, ainsi que les **fichiers de configuration JSON** utilisés pour définir les paramètres initiaux de la ville (nombre de parkings, capacités, etc.

***Build:*** Ce dossier est destiné à recevoir les fichiers générés lors de la compilation. Il permet de garder le répertoire racine propre en isolant les fichiers objets et l'exécutable final.

***Include:*** Ce dossier regroupe tous les **fichiers d'en-tête** . C'est ici que sont définies les interfaces des classes , garantissant une bonne encapsulation et la visibilité des structures de données à travers le projet.

***Src:*** Il contient le **code source (.cpp)**. C'est ici que réside la logique métier de votre simulateur de parking, notamment l'implémentation

des méthodes de classes et les algorithmes de choix de stationnement.

***Tests:*** Conformément aux exigences du projet, ce dossier est dédié aux **tests unitaires**. Il contient les scripts permettant de vérifier le bon fonctionnement des classes principales (au moins 5 tests sont requis par le sujet)

## *Fichiers racines:*

- **`CMakeLists.txt`** : C'est le fichier de configuration pour le système de construction CMake. Il définit comment compiler le code C++, comment lier la bibliothèque **Raylib** et comment gérer les dépendances du projet.
- **`LICENSE`** : Ce document précise les termes juridiques et les droits d'utilisation du code source produit.
- **`README.md`** : Il s'agit de la documentation principale du projet. Ce fichier explique comment installer, compiler et lancer le simulateur, tout en fournissant un aperçu des fonctionnalités implémentées.

**On a separé le jeu en plusieurs classes :**

### *Parkinglogic.hpp:*

C'est le cœur métier de votre sous-projet. Conformément aux consignes sur le **Smart Parking**, ce fichier définit la logique de gestion des espaces de stationnement.

- **Rôle :** Gérer la capacité des parkings et le taux d'occupation en temps réel.

- **Fonctionnalités :** * Implémente les **stratégies de recherche** (choix du parking le plus proche ou le moins cher).
- Gère la **résolution de conflits**, par exemple lorsque deux véhicules ciblent la même place au même moment.

## ***Carlogic.hpp:***

Ce fichier contient la définition des véhicules et de leurs comportements individuels.

- **Rôle :** Modéliser les véhicules circulant dans la ville et cherchant à se garer.
- **Fonctionnalités :** * Définit les attributs physiques (vitesse, position) et les états du véhicule (en mouvement, en attente, garé).
- Gère l'interaction avec le système de parking pour demander une place libre.

## ***Simulation.hpp:***

Il s'agit du chef d'orchestre de votre application. Ce fichier centralise la boucle principale du simulateur.

- **Rôle :** Coordonner l'initialisation de **Raylib** et la mise à jour de tous les modules.
- **Fonctionnalités :** * Gère le "tick" de simulation (mise à jour des positions des voitures et de l'état des parkings).
- S'occupe du rendu visuel global (affichage 2D ou 3D des infrastructures).

## ***Component.hpp:***

Ce fichier regroupe les petites structures de données ou les objets réutilisables par plusieurs classes.

- **Rôle :** Fournir des briques de base pour construire les objets plus complexes de la Smart City.
- **Fonctionnalités :** * Contient généralement des structures comme `Position`, `Dimension` ou des éléments d'interface graphique (UI) pour le tableau de bord de suivi.

## ***Utils.hpp:***

Comme son nom l'indique, ce fichier contient les outils utilitaires et les fonctions d'aide.

- **Rôle :** Centraliser les fonctions mathématiques, les constantes globales ou les outils de lecture de fichiers.
- **Fonctionnalités :** * Peut inclure les fonctions de calcul de distance entre un véhicule et un parking.
- Gère souvent le chargement de la configuration **JSON** pour initialiser les routes et les capacités des parkings.

# ***Les Design Patterns:***

| | |
|---|---|
| **Game Loop** | Maintient la simulation vivante à 60 FPS. |
| **State Pattern** | Gère le passage de l'image de titre au dashboard. |
| **Update Method** | Calcule la nouvelle position des 50 voitures à chaque frame. |
| **Data-Driven** | Permet de configurer les prix et couleurs des parkings facilement. |

## ***1.game loop:***

C'est le pattern le plus évident. Au lieu d'attendre une action utilisateur pour rafraîchir l'écran, notre programme tourne en boucle infinie jusqu'à la fermeture.

• **Où dans notre code :** Les boucles while (!WindowShouldClose()).

• **Utilité :** Elle synchronise la mise à jour de la logique (UpdateTraffic) et le rendu graphique (BeginDrawing)

## 2.state pattern:

Gére la navigation entre l'écran d'accueil, les infos et la simulation via une machine à états simple.

• **Où dans notre code :** L'utilisation de l'énumération enum class ScreenState { INTRO, INFO, SIM }; et la variable state.

• **Utilité :** Cela permet de changer radicalement le comportement du programme (ce qu'on dessine et ce qu'on calcule) selon l'état actuel.

## 3.update method:

Plutôt que de tout coder dans le main,nous déléguons la logique à des fonctions spécialisées qui reçoivent le "delta time" (dt).

• **Où dans notre code :** UpdateTraffic(cars, roads, parkings, dt);.

• **Utilité :** Cela permet de simuler un mouvement fluide indépendant de la puissance de l'ordinateur.

## 4.strategy pattern:

La structure de nos voitures et la manière dont elles choisissent leur comportement (conduire, chercher une place, se garer) s'apparente à ce pattern.

• **Où dans notre code :** Le champ c.state = DRIVING; dans la structure Car.

• **Utilité** : Selon cet état, l'algorithme de mouvement change.

## 5.data-driven method:

Vous ne créez pas chaque parking à la main avec du code dupliqué. Vous utilisez des structures de données et des boucles pour les générer et les afficher.

• **Où dans notre code :** Les vecteurs parkingPositions, parkingPrices et la boucle for (const auto& p : parkings) DrawParking(p);.

• **Utilité :** Si nous voulons ajouter 10 parkings de plus, nous n'avons qu'à ajouter une ligne dans le vecteur, sans changer la logique d'affichage.

# 3.fonctionnalités principales:

Pour implémenter les fonctionnalités principales du jeu, nous avons structuré le développement en plusieurs étapes clés. Voici comment chaque fonctionnalité a été conçue et intégrée:

## CAR HPP

Voici une explication du code fourni :

### key point:

- La ligne `#pragma once` empêche les inclusions multiples du même fichier d'en-tête lors de la compilation.

- L'inclusion de `Components.hpp` importe les définitions nécessaires (probablement pour `Car` et `Road`).

- La fonction `DrawCar(const Car& car, const Road& road)` dessine une voiture avec une orientation qui dépend de sa direction sur la route ou dans le parking.

### Détails sur la fonction :

### Paramètres :

- `const Car& car` : référence constante au modèle de voiture à dessiner.

- `const Road& road` : référence constante à la route où la voiture est placée.

But :

- Dessiner la voiture (graphiquement, probablement sur une interface ou une simulation).

- Adapter l'orientation de la voiture en tenant compte de sa direction dans le contexte (route ou parking).

**Remarque :** Le corps de la fonction n'est pas fourni, il s'agit donc seulement de sa déclaration.

## Résumé du code :

- La ligne `#pragma once` évite que ce fichier soit inclus plusieurs fois lors de la compilation, ce qui prévient les erreurs de double déclaration.

- La ligne `#include "Components.hpp"` importe les définitions des éléments nécessaires, probablement les classes/structs `Car` (voiture) et `Road` (route).

- La déclaration `void DrawCar(const Car& car, const Road& road);` indique qu'il y a une fonction appelée DrawCar qui :

- Prend en entrée une voiture (`car`) et une route (`road`), toutes deux transmises par référence constante (ce qui évite la copie et protège les données d'une modification).

- Va dessiner la voiture selon sa direction actuelle, c'est-à-dire si elle est sur la route ou dans un parking.

**En résumé :** Ce code annonce une fonction (sans l'implémentation) qui sert à dessiner une voiture en tenant compte de son orientation sur la route ou dans un parking. Elle s'appuie sur des composantes définies ailleurs (dans `Components.hpp`).

## CAR .CPP

Voici une explication de la fonction DrawCar :

**Points clés :**

***1. Position et orientation de la voiture***

**a.** En conduite : La position est calculée en suivant la direction de la route avec un décalage latéral (laneOffset). L'angle de rotation vient de la direction de la route.

**b.** À l'arrêt : On utilise la position réelle (worldPos), et la voiture tourne vers sa cible (targetPos). Si elle est très proche, l'angle est fixé à 90° (vertical).

***2. Dessiner les roues***

**a.** Les 4 roues sont dessinées grâce à des offsets. Chaque roue est placée via DrawRectanglePro pour le bon positionnement et la rotation.

***3. Corps de la voiture***

**a.** Le rectangle principal représente le châssis, centré sur sa position, et tourné selon l'angle calculé.

***4. Contour de la voiture***

**a.** Comme DrawRectangleLinesPro pose problème, tu utilises un polygone 4 côtés (DrawPolyLinesEx) pour dessiner le contour en rotation.

***5. Vitre / pare-brise***

**a.** Un petit rectangle plus sombre (verre) est dessiné en haut de la voiture.

***6. Phare avant (jaune) & feu arrière (rouge)***

**a.** Deux cercles jaunes à l'avant pour les phares, deux rouges à l'arrière pour les feux de recul.

## COMPNEMENT.HPP

- **Constantes importantes**

IL centralise les dimensions de la voiture, la distance de sécurité et la vitesse maximale dans des constantes, ce qui améliore la maintenabilité du code.

- **Enums clairs**

Les énumérations (`LightState` pour les feux, `CarState` pour l'état de la voiture) rendent la logique des états beaucoup plus lisible et robuste.

- **Struct TrafficLight**

Gère simplement la gestion temporelle et l'état du feu, avec une position fixe sur la route.

- **Struct Road**

Comprend le départ, l'arrivée, le nombre de voies, la largeur, et inclut directement un feu.

Méthodes utiles pour obtenir la longueur (`getLength`) et la direction normalisée (`getDir`).

- **Struct Car**

Toutes les informations nécessaires à la simulation : position, vitesse, voie, état, couleur.

Constructeur propre : tous les champs sont initialisés à des valeurs par défaut raisonnables.

Gère le changement de voie, le stationnement, la file d'attente et la cible.

- **Struct ParkingLot**

Contient la position, taille, capacité, prix du parking, nom, couleur, position de sortie, et l'état d'occupation des places sous forme de vector de booléens.

### - Méthodes utiles :

`firstFreeSpot` pour trouver la prochaine place libre,

`occupySpot` / `freeSpot` pour gérer les entrées/sorties.

Constructeur explicite, pratique pour créer des parkings différents.

## Utile.hpp

**Fonctions déclarées :**

DrawDashedLine :

Prend deux points (start, end), une épaisseur (thickness) et une couleur (Color color).

Sert à dessiner une ligne pointillée entre deux points à l'écran.

**DrawDriveway :**

Prend une référence vers un ParkingLot et une Road.

Sert à dessiner l'allée entre le parking et la route principale.

**Dépendances :**

- Inclut raylib.h (pour le dessin graphique).

- Inclut Components.hpp

- Inclut Simulation.hpp

## Utile cpp

Voici ce que fait le code :

**DrawDashedLine**

Dessine une ligne pointillée entre deux points (start, end).

Utilise un intervalle: tiret de 20 pixels suivi d'espace de 20 pixels (40 px par « segment »).

Calcule la direction unitaire du segment avec raymath.

- Pour chaque segment :

Dessine un trait de 20 pixels de long dans la bonne direction.

Le dernier trait s'ajuste si la distance restante est < 20 px.

**DrawDriveway**

Dessine l'allée entre un ParkingLot et une Road :

Calcule le centre X du parking.

Prend le bord du parking « le plus proche » de la route selon la position Y.

Génère deux points :

Un sur la route (roadPoint),

Un sur le parking (parkingPoint).

Trace :

Une large ligne foncée (140px) pour l'allée centrale.

Deux lignes grises fines (2px) à 20px à gauche et à droite de l'allée (marquages ou bordures).

**Résumé des points clés :**

DrawDashedLine crée une ligne pointillée entre deux coordonnées selon une pattern fixe.

DrawDriveway relie l’entrée du parking à la route principale visuellement, avec marquage central et bordures.

**CMakeLists.txt** est la "recette" qui permet de transformer tes fichiers sources en programmes utilisables. Voici l'explication détaillée de ce qu'il fait, étape par étape :

## 1. Initialisation et Standards:

• **cmake_minimum_required :** Il impose l'utilisation de CMake version 3.14 ou plus pour garantir la compatibilité des commandes utilisées.

• **project :** Il donne un nom au projet (SmartCitySim) et précise qu'il utilise le langage C++ (CXX).

•**set(CMAKE_CXX_STANDARD 17) :** Il configure le compilateur pour utiliser la norme C++17, ce qui permet d'utiliser des fonctionnalités modernes comme les std::vector avancés ou certaines fonctions mathématiques.

## 2. Gestion des Chemins(Path):

C'est ici que CMake localise les outils nécessaires :

• **RAYLIB_PATH :** C'est une variable qui pointe vers l'endroit où tu as installé la bibliothèque graphique Raylib.

• **include_directories :** Il indique au compilateur où chercher les fichiers .hpp (tes headers et ceux de Raylib).

• **link_directories :** Il précise où se trouvent les fichiers binaires de Raylib (.lib ou .a) nécessaires pour l'étape finale de création de l'exécutable.

## 3. Création de l'application principale (SmartCitySim):

• **add_executable(... WIN32 ...) :** Cette commande crée l'application que tu vois dans la vidéo. L'option WIN32 est cruciale car elle indique à Windows de lancer le programme en "Mode Fenêtre" (sans ouvrir une console noire derrière).

• **Fichiers sources :** Il rassemble tous les morceaux du puzzle : main.cpp (l'interface), Simulation.cpp (le moteur de trafic), ParkingLogic.cpp (la gestion des places), etc..

• **target_link_libraries :** Il "attache" les bibliothèques système de Windows (gdi32 pour le graphisme, winmm pour le son) à ton programme pour qu'il puisse afficher les voitures et jouer les bruits de moteur.

## 4. Création du programme de Tests (TrafficTests):

• **add_executable(TrafficTests ...) :** Il crée un deuxième programme, plus léger, qui sert uniquement à vérifier que ton code fonctionne correctement.

• **set_target_properties ... -mconsole :** Contrairement à l'application principale, ce programme est forcé de s'ouvrir dans une Console. C'est pratique pour lire des messages d'erreur ou des résultats de tests textuels sans lancer toute l'interface graphique.

**Pour les constructeurs el les destructeurs on a :**

## Les constructeurs

**1. Constructeur de la structure Car**

Dans components.hpp on trouve:

Pour la voiture, le constructeur permet de définir une identité (ID) et une position de départ, tout en mettant tous les autres paramètres (vitesse, timers, états) à zéro ou à une valeur par défaut sécurisée.

**2. Constructeur de la structure ParkingLot**

Pour le parking, le constructeur est plus dynamique car il doit dimensionner le vecteur spotsOccupied en fonction de la capacité fournie.

## Les destructeurs

Assure un nettoyage éventuel des ressources (non implémenté ici, mais pourrait inclure UnloadTexture(), UnloadMusicStream(menuMusic); CloseAudioDevice() ).

**Et pour le ressources :**

## Structure des Ressources:

• **assets/accueil.png :** Image d'introduction.

• **assets/menu.mp3 :** Musique de fond.

## Main.cpp

## *1. Fonctions de Logique Géographique et Calcul*

• Ces fonctions traitent les données mathématiques pour donner du sens aux coordonnées spatiales.

• **GetCardinalSide :** Calcule la position relative d'un point (un parking) par rapport au centre de la ville. Elle compare l'écart en X et en Y pour déterminer si le parking est plutôt au Nord, Sud, Est ou Ouest.

• **Clamp01 :** Une fonction de sécurité utilitaire qui force une valeur à rester entre 0.0 et 1.0. Elle est indispensable pour la gestion du volume sonore afin d'éviter des erreurs de calcul audio.

## *2. Fonctions d'Interface Utilisateur (UI)*

Ces fonctions gèrent l'affichage visuel et l'interaction avec l'utilisateur.

• **DrawIntroFullScreen :** Gère l'affichage de l'image de démarrage (accueil.png). Elle utilise DrawTexturePro pour s'assurer que l'image s'adapte parfaitement à la taille de votre fenêtre (1200x900), même si l'image d'origine a des dimensions différentes.

• **DrawInfoSheet :** Dessine le tableau central bordeaux qui récapitule les tarifs. Elle parcourt votre vecteur parkingInfos pour aligner proprement le nom, le prix et le côté cardinal de chaque parking.

• **UpdateAudioUI & DrawAudioUI :** Ce binôme gère le panneau audio. La première détecte si la souris déplace le curseur (slider) ou clique sur "Muet", tandis que la seconde dessine les rectangles et le bouton vert/rouge correspondant à l'état du son.

• **DrawParkingDashboard :** C'est le centre de contrôle visuel en haut de l'écran. Elle transforme les données d'occupation en barres de progression visuelles.

## *3. Fonctions de Logique de Simulation*

• Elles représentent le "moteur" de notre ville intelligente.

• **CountOccupiedSpots :** Une fonction simple mais cruciale qui parcourt le vecteur de booléens (spotsOccupied) d'un parking pour compter combien de places sont à true.

• **UpdateTraffic (Appelée dans le main) :** Cette fonction (définie dans nos fichiers externes) est le cœur du projet. Elle met à jour la position de chaque voiture, vérifie si le feu est rouge, et décide si une voiture doit entrer dans un parking ou continuer sa route.

• **DrawCar :** Calcule la position réelle à l'écran d'un véhicule en fonction de sa distance sur la route et de sa file (Lane), puis dessine un rectangle coloré orienté dans la bonne direction.

## *4. La Boucle Principale (main)*

Le main orchestre la transition entre les différents états du programme :

• **Phase d'Initialisation :** Chargement des textures, de la musique et création des objets Road, ParkingLot et Car.

• **Machine à États (ScreenState) :**

- **INTRO :** Attend un appui sur ESPACE.
- **INFO :** Affiche les tarifs et permet de régler le son.
- **SIM :** Lance le moteur physique et le chronomètre.

• **Nettoyage :** Avant de fermer la fenêtre, elle décharge la mémoire (UnloadTexture, UnloadMusicStream) pour éviter que le programme ne laisse des traces dans la RAM.

# Simulation.hpp

Voici ce que fait ce code header :

Inclut les dépendances nécessaires : Il importe "Components.hpp" et les headers de la STL utiles.

Déclare deux fonctions majeures pour la simulation du trafic :

**- IsLaneFree :**

Vérifie si une voie donnée est libre pour un changement de voie d'une voiture.

Prend en paramètres la liste des voitures, l'index de la route, la voie à vérifier, la position de la voiture et son ID.

Retourne un booléen (vrai si la voie est libre).

**- UpdateTraffic** :

Met à jour la totalité du trafic, y compris les parkings.

Modifie les listes de voitures, routes et parkings en fonction de l'évolution temporelle (dt).

# Simulation.cpp

## *1. Le Gestionnaire des Feux (TrafficLight::update)*

Cette fonction gère le passage automatique entre les trois états du feu tricolore.

• **Logique temporelle :** Elle utilise un timer qui diminue à chaque seconde. Quand il atteint zéro, l'état change (Vert rightarrow Jaune rightarrow Rouge).

• **Durées :** Le cycle est fixé à 5s pour le vert et le rouge, et 2s pour le jaune pour simuler un comportement réaliste.

## *2. Évitement de Collisions (IsLaneFree)*

C'est la fonction "sécurité". Avant qu'une voiture ne change de voie, elle scanne l'environnement :

• **Filtrage :** Elle ignore la voiture actuelle et les voitures sur d'autres routes.

• **Calcul de distance :** Si une autre voiture sur la voie cible se trouve à moins de 1.5 la SAFE_DISTANCE, elle renvoie false. Cela empêche les voitures de se téléporter les unes sur les autres.

## *3. Le Moteur de Trafic (UpdateTraffic)*

C'est la fonction la plus complexe. Elle se divise en quatre étapes :

### A. Tri pour la Cohérence

Notre code utilise std::sort pour classer les voitures de la plus avancée à la plus reculée.

**Pourquoi ?** Pour qu'une voiture sache toujours ce que fait celle qui est devant elle avant de calculer son propre mouvement.

### B. Prise de Décision (IA)

Chaque voiture a une chance aléatoire (GetRandomValue(0, 500) < 2) de vouloir se garer.

Si elle décide de le faire :

**1.** Elle cherche le parking le plus proche avec une place libre.

**2.** Elle vérifie si elle est sur la bonne voie (les parkings du bas sont accessibles par la voie 0, ceux du haut par la voie 1).

### C. Gestion de la Vitesse (ACC - Adaptive Cruise Control)

Le code calcule une distToObstacle en comparant :

• La voiture de devant.

• L'état du feu tricolore (considéré comme un obstacle s'il est rouge).

La vitesse est ensuite ajustée via une interpolation linéaire (Lerp) pour un freinage et une accélération fluides.

### *4. La Machine à États des Véhicules*

Une voiture passe par plusieurs états gérés dans la boucle finale :

**1. DRIVING :** La voiture suit la route.

**2. TO_PARKING :** Elle quitte la route pour rejoindre sa place. Le code gère ici une "file d'attente" pour éviter que deux voitures ne se chevauchent dans l'allée du parking.

**3. PARKED :** Elle attend (état d'arrêt total).

**4. LEAVING_PARKING :** Elle se dirige vers exitPos. Une fois arrivée, elle libère la place dans la structure ParkingLot (freeSpot) et redevient DRIVING.

## *Synthèse des calculs mathématiques utilisés*

• **Vecteurs :** Utilisation de Vector2Distance pour la proximité et Vector2Normalize pour la direction du mouvement.

• **Normales :** Le calcul normal = { -dir.y, dir.x } permet de décaler les voitures perpendiculairement à la route pour créer des voies (Lanes)

# Parkinglogic.hpp

### *1. UpdateParking(ParkingLot& p)*

cette fonction est destinée à la maintenance du parking :

• **Gestion du temps :** Elle pourrait servir à calculer les revenus générés en fonction du temps d'occupation.

• **Animation**

### *2. DrawParking(const ParkingLot& p)*

C'est la fonction graphique principale. Elle utilise les données de la structure ParkingLot pour :

• **Tracer le rectangle** représentant la zone de stationnement avec sa couleur spécifique (Bleu pour VIP, Vert pour Eco, etc.).

• **Dessiner les places :** Elle boucle sur le nombre de places et dessine un indicateur (souvent un rectangle vide ou une voiture miniature) selon que la place est libre ou occupée.

• **Afficher le texte :** Elle écrit le nom du parking et son prix au-dessus de la zone.

### *3. GetSpotPosition(const ParkingLot& p, int spotIndex)*

C'est une fonction de calcul géométrique indispensable pour la navigation des voitures.

• **Entrée :** L'objet parking et le numéro de la place (ex: place n°3).

• **Sortie :** Un Vector2 (coordonnées x, y ) exact où la voiture doit s'arrêter.

• **Calcul interne :** Elle répartit les places à l'intérieur du rectangle du parking. Par exemple, elle divise la largeur totale par le nombre de places pour que les voitures soient bien alignées et ne se chevauchent pas.

## Parkinglogic.cpp

### *1. GetSpotPosition : Le Calculateur de Coordonnées*

Cette fonction est le "GPS" interne du parking. Elle ne dessine rien, mais elle calcule l'endroit exact où une voiture doit s'arrêter.

• **Logique de Grille (Matrix logic) :** Pour éviter que toutes les voitures s'empilent au même endroit, le code calcule un nombre de colonnes (cols) en fonction de la largeur du parking.

• **Modulo et Division :**

o **row = spotIndex / cols :** Détermine l'étage (ligne).

o **col = spotIndex % cols :** Détermine la position latérale (colonne).

• **Point Pivot :** La fonction renvoie le centre de la place (x + width/2, y + height/2), ce qui permet à la voiture de se garer pile au milieu du rectangle.

## *2. DrawParking : Le Rendu Graphique*

Cette fonction s'occupe de tout l'aspect visuel en trois couches :

### *A. La Structure de base (Bitume)*

• Elle dessine le sol du parking en gris foncé (0x2A2A2AFF) et ajoute une bordure blanche pour bien délimiter la zone de stationnement par rapport à l'herbe.

### *B. Le Panneau d'Information*

• Un petit rectangle de couleur (propre à chaque parking) est dessiné juste audessus.

• Il affiche le Nom (ex: VIP) et le Prix (ex: 15dh/h) en utilisant TextFormat pour intégrer la variable numérique dans le texte.

### *C. La Gestion des Places (Indicateurs d'occupation)*

Le code boucle sur la capacité du parking pour dessiner chaque emplacement :

• **Calcul dynamique :** Comme dans GetSpotPosition, il recalcule la position x,y de chaque emplacement.

• **État visuel :**

o **Si occupé (p.spotsOccupied[i]) :** Dessine un rectangle Rouge (représentant une voiture garée)

. o **Si libre :** Dessine un rectangle Gris foncé (place vide).

• **Sécurité :** Le code vérifie if (y + spotHeight > p.position.y + p.size.y) pour s'assurer que les places ne dépassent pas physiquement du rectangle du parking si la capacité est trop grande.

## Tests/testraffic.cpp

Un test unitaire est un procédé de programmation permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'un programme.

On distingue entre trois tests:

### *1. Test du Feu Rouge (TestFeuRouge)*

Ce test vérifie si le véhicule adapte son comportement face à une signalisation restrictive.

• **Objectif :** S'assurer que la voiture freine lorsqu'elle approche d'un feu rouge.

• **Fonctionnement :** Nous avons placé une voiture à une distance stratégique (460 pixels) sur une route de 500 pixels équipée d'un feu rouge.

• **Résultat validé :** La vitesse est passée de 100 à 0, prouvant que l'algorithme de détection de distance (distToLight) et le freinage d'urgence fonctionnent parfaitement.

### *2. Test de Collision (TestCollision)*

Ce test valide l'interaction entre les agents (véhicules) sur la route.

• **Objectif :** Vérifier que deux voitures ne se rentrent pas dedans.

• **Fonctionnement :** On place deux voitures sur la même voie, l'une derrière l'autre, avec une vitesse élevée pour la seconde.

• **Résultat validé :** Le test affiche [OK] car la voiture de derrière détecte l'obstacle (distToObstacle) et réduit sa vitesse pour maintenir une distance de sécurité.

### *3. Test d'Entrée au Parking (TestEntreeParking)*

Ce test vérifie la logique de transition d'état et la navigation vers les zones de stationnement.

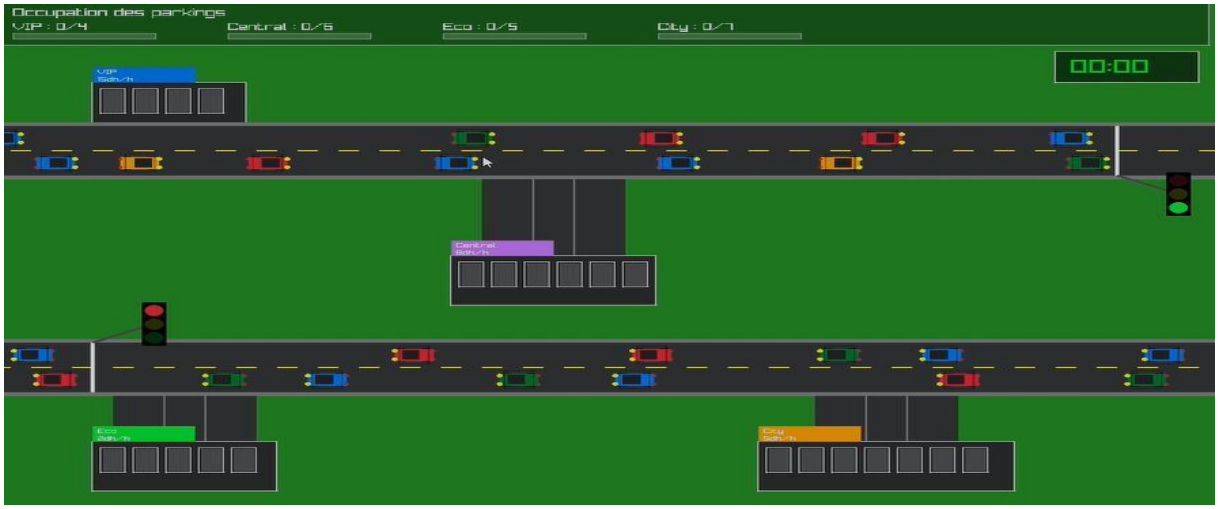
• **Objectif :** Confirmer qu'une voiture peut changer son mode de conduite pour aller se garer.

• **Fonctionnement :** Une voiture est placée sur la bonne voie (lane 1 pour les parkings hauts) à proximité d'un parking ayant au moins une place libre (spotsOccupied = {false}).

• **Résultat validé :** L'état de la voiture est passé avec succès de DRIVING à TO_PARKING, validant ainsi le système de réservation de place et la détection des zones d'entrée.

# 4. captures et scenario:

### Captures d’écran du jeu


```
C:\Users\hp\Documents\SmartCity>.\build\TrafficTests.exe
===== LANCEMENT DES TESTS =====
[OK] Test Feu Rouge Reussi. Vitesse : 0
[OK] Test Parking Reussi.
[OK] Test Collision Reussi.
===== TOUS LES TESTS SONT FINIS =====
```


# Scenario de simulation

## 1. Phase d'Initialisation et Menu

• **Écran d'accueil :** La simulation s'ouvre sur un menu intitulé "Parking Fury" avec une musique de moteur en arrière-plan.

• **Configuration :** L'utilisateur peut régler le volume sonore via un curseur en haut à gauche.

• **Tableau d'information :** Un panneau central affiche les tarifs horaires des quatre types de parkings : VIP (15 dh/h), Central (8 dh/h), Eco (2 dh/h) et City (5 dh/h), ainsi que leur localisation géographique

## 2. Lancement et Environnement

• **Démarrage :** En cliquant sur "Démarrer", l'interface de simulation apparaît.

• **Structure de la Map :** La scène est composée de deux routes principales à double sens, séparées par un terre-plein central vert.

• **Apparition du trafic :** Des voitures de couleurs variées (rouge, bleu, vert, jaune) apparaissent sur la gauche et circulent vers la droite.

## 3. Logique de Stationnement et Sélection

• **Recherche de place :** Les véhicules circulent sur la route jusqu'à ce qu'ils décident de stationner. Le système vérifie la disponibilité en temps réel, affichée dans les compteurs en haut de l'écran (ex: "VIP : 0/4", "Central : 0/5").

• **Sélection intelligente :** Les voitures se dirigent vers le parking le plus proche de leur position actuelle, tout en évitant les zones restreintes si elles ne sont pas autorisées.

• **Manoeuvre d'entrée :** Une fois devant un parking, la voiture effectue une rotation fluide pour s'aligner avec une place libre et s'y immobilise.

### *4. Gestion du Trafic et Sécurité*

• **Feux tricolores :** Des feux de signalisation aux extrémités des routes régulent le flux. Les voitures s'arrêtent au rouge et repartent au vert.

• **Radar Anti-collision :** Grâce au code de simulation, chaque voiture maintient une distance de sécurité. Si une voiture devant s'arrête (au feu ou pour tourner), celle de derrière freine automatiquement pour éviter l'accident.

### *5. Cycle de Sortie (Le "Leaving")*

• **Temps d'attente :** Chaque voiture reste garée pendant une durée aléatoire définie par un waitTimer.

• **Sortie par la voie dédiée :** Au lieu de disparaître, la voiture quitte sa place, s'oriente vers la petite route de sortie (la bande grise que tu as ajoutée) et rejoint la circulation principale de manière réaliste.

• **Libération de place :** Dès que la voiture quitte le parking, le compteur d'occupation diminue instantanément, permettant à un autre véhicule d'entrer.

### *6. Fin de Cycle et Boucle Infinie*

• **Recyclage des entités :** Lorsqu'une voiture sort de l'écran à droite, elle est supprimée et une nouvelle est générée à gauche pour maintenir un flux continu.

• **Horloge :** Un chronomètre en haut à droite suit la durée totale de la simulation

# *5.conclusion:*

Au terme de ce projet, nous avons pu démontrer l’importance et la pertinence des solutions de mobilité intelligente au sein des villes modernes. L’étude et la simulation du Smart Parking réalisées dans le cadre de ce travail illustrent comment des technologies numériques, combinées à des modèles de décision et à une

architecture logicielle cohérente, peuvent contribuer à améliorer la gestion urbaine en réduisant la congestion, le temps de recherche de stationnement et les émissions polluantes.

L'implémentation du système en C++ à l'aide de la bibliothèque Raylib nous a permis de concevoir une simulation dynamique intégrant plusieurs parkings, un flux continu de véhicules, un tableau de bord informatif et une logique de réservation et de libération de places. Ce travail nous a également permis de mettre en pratique des concepts essentiels de la programmation orientée objet, de l'architecture logicielle, de la simulation temps réel et de l'utilisation de design patterns pour organiser un ensemble d'interactions complexes. Le projet a ainsi montré qu'un environnement simulé constitue un outil efficace pour tester des mécanismes intelligents avant une éventuelle implémentation réelle.

Bien que le système développé réponde à un ensemble d'exigences fonctionnelles liées au Smart Parking, plusieurs extensions demeurent possibles et pertinentes. Parmi celles-ci, l'intégration de la recharge intelligente des véhicules électriques (EV Charging), la prise en compte de contraintes énergétiques, la modélisation de comportements prédictifs, l'ajout d'algorithmes d'optimisation du stationnement ou encore la communication inter-parkings représentent des pistes d'amélioration prometteuses. L'extension de la simulation au niveau de la ville entière ou l'ajout de données réelles issues de capteurs IoT pourraient également renforcer la fidélité du système et améliorer son utilité opérationnelle.

En conclusion, ce projet a permis de mieux comprendre les enjeux techniques et urbains liés à la Smart City, tout en confirmant le rôle crucial des solutions logicielles et des simulations dans l'analyse, l'expérimentation et la validation de concepts innovants. Au-delà de son aspect académique, ce travail ouvre des perspectives intéressantes pour la recherche et le développement de futures applications dédiées à la gestion intelligente de la mobilité urbaine.