

# Rapport de Projet – Smart city

## Sous-projet 5 :

## Emergency & Priority Management

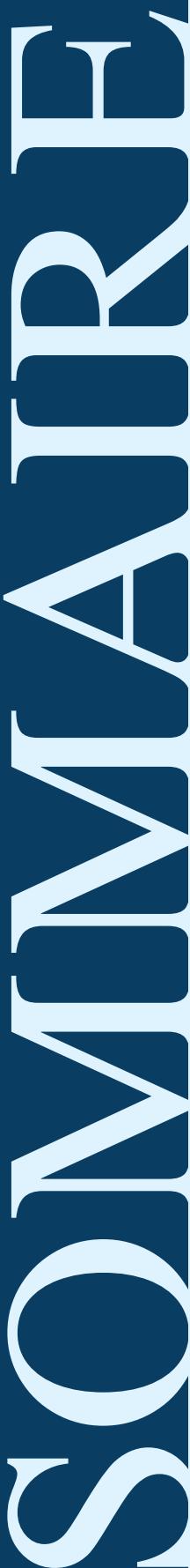


**Encadré par :**

**Réalisé par :**

Pr.Ikram Benabdelouahab

- Oussama El Boualiti
- Zaid Chahid
- Ikram Houssas
- Farah Belfaquihi
- Salma Samah



- 1. Introduction...p03**
  - 1.1 Contexte du projet
  - 1.2 Objectifs du sous-projet 5
  - 1.3 Périmètre technique
- 2. Architecture du Logiciel et Modélisation POO...p04**
  - 2.1 Hiérarchie des classes (Héritage)
  - 2.2 Encapsulation et visibilité
  - 2.3 Polymorphisme et fonctions virtuelles
- 3. Conception des Design Patterns...p08**
  - 3.1 Pattern 1 : State (gestion des états du véhicule)
  - 3.2 Pattern 2 : Strategy (planificateur de trajets)
- 4. Implémentation des Fonctionnalités Clés...p11**
  - 4.1 Gestion de la simulation et utilisation de la STL
  - 4.2 Interaction Véhicules-Feux (Couloirs Verts)
  - 4.3 Planificateur de trajets optimisés
  - 4.4 Gestion des exceptions
- 5. Visualisation avec Raylib...p21**
  - 5.1 Représentation graphique des véhicules prioritaires
  - 5.2 Interface utilisateur et missions en temps réel
- 6. Tests Unitaires et Validation...p25**
  - 6.1 Présentation des tests unitaires
- 7. Scénario de Démonstration...p30**
  - 7.1 Étapes de la démo : ambulance, feux adaptatifs, comportement des véhicules normaux
- 8. Conclusion et Perspectives...p35**
  - 8.1 Synthèse du travail réalisé
  - 8.2 Ouverture sur l'intégration avec les autres modules
- 9. Annexes...p40**
  - 9.1 Diagramme UML détaillé
  - 9.2 Interfaces Raylib
  - 9.3 Liens du code source et de la vidéo

## 1. Introduction

Dans une Smart City, il est important de gérer correctement les situations d'urgence comme les accidents de la route. Les ambulances et les véhicules de dépannage doivent pouvoir circuler rapidement, même lorsque le trafic est dense, afin de réduire les délais d'intervention et d'améliorer la sécurité des citoyens.

Ce sous-projet a pour but de créer une simulation simple et réaliste de la gestion des véhicules en général et surtout prioritaires dans un trafic urbain. Le projet est réalisé en C++ en utilisant la programmation orientée objet, avec la bibliothèque Raylib pour l'affichage graphique

## 2. Objectifs du projet

Les objectifs principaux sont :

- Simuler un trafic routier avec plusieurs voies
- Créer des accidents de manière dynamique
- Gérer l'intervention d'une ambulance
- Gérer l'intervention d'un véhicule de dépannage
- Donner la priorité aux véhicules d'urgence
- Intégrer un bus pour le transport des étudiants vers l'école.
- Observer le comportement des autres véhicules
- Afficher clairement les événements à l'écran

## 1.3 Périmètre technique

- **Langage** : C++ (Standards C++14/17).
- **Bibliothèque Graphique** : [Raylib](#) (Choisie pour sa légèreté et sa gestion efficace du rendu 2D et de l'audio).
- **Architecture** : Boucle de jeu (Game Loop) avec mise à jour logique (Update) et rendu graphique (Draw) séparés.
- **Gestion Mémoire** : Utilisation exclusive de pointeurs intelligents (std::unique\_ptr) pour garantir l'absence de fuites de mémoire (RAII).

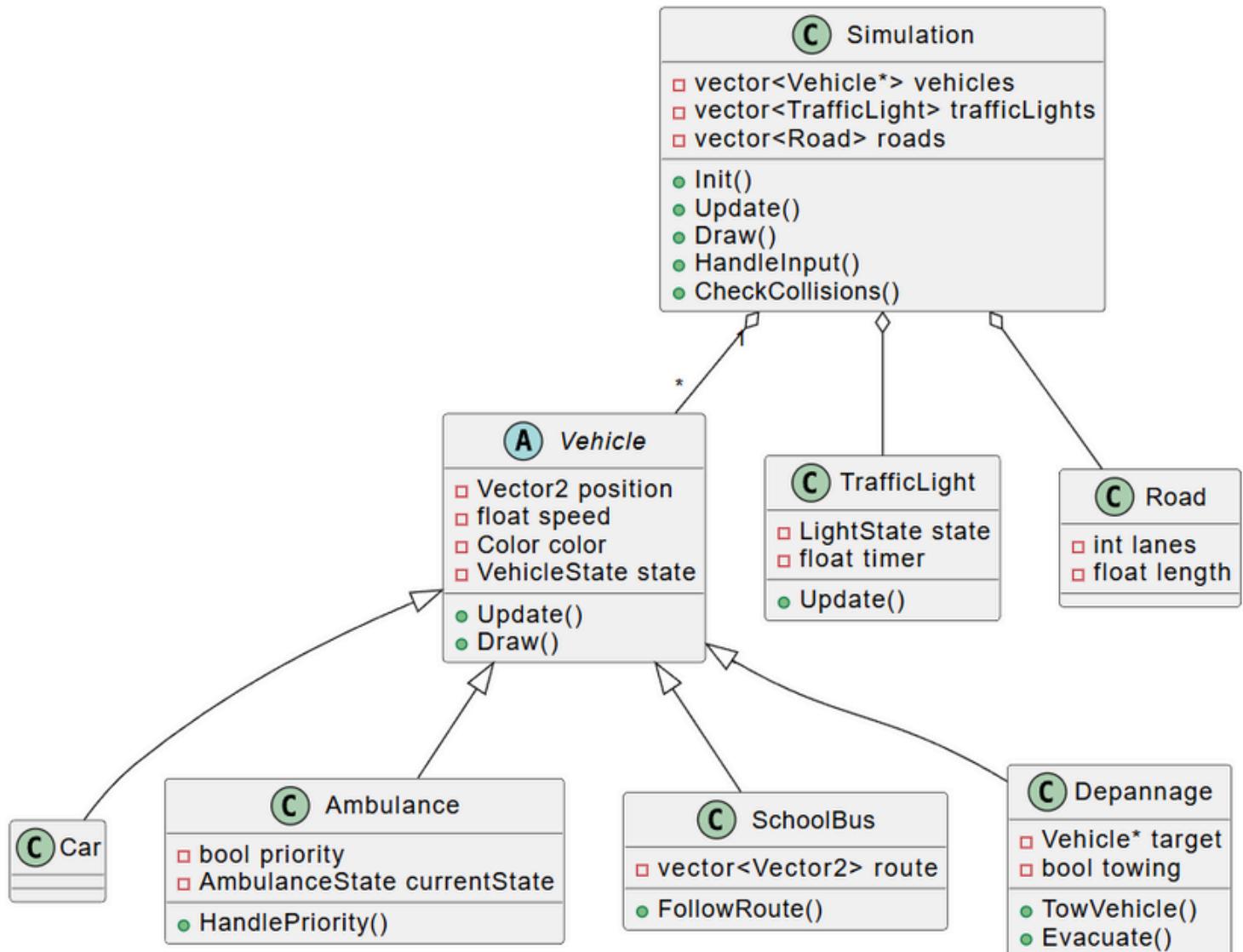
## 2. Architecture du logiciel et POO

L'architecture repose sur un modèle hiérarchique utilisant l'héritage et le polymorphisme pour gérer les différents types d'entités.

### 2.1 Hiérarchie & Diagramme de Classes

1. **Simulation (Le Moteur)**
  - Contient les conteneurs de véhicules (std::vector).
  - Gère la logique globale (collisions, apparitions, entrées clavier).
  - Gère la caméra et l'interface utilisateur (UI).
2. **Vehicle (Classe Mère / Base)**
  - Attributs : Position (x, y), Vitesse, Couleur, État (crashé, remorqué).
  - Méthodes Virtuelles : Update() (déplacement) et Draw() (rendu).
  - Rôle : Définit le comportement par défaut d'un véhicule (avancer, s'arrêter si obstacle).
3. **Classes Dérivées (Spécialisation)**
  - **Car** : Véhicule standard.
  - **Ambulance** : Comportement prioritaire (ignore les feux rouges), machine à états (vers accident, vers hôpital).
  - **SchoolBus** : Itinéraire scripté (arrêt spécifique à l'école).
  - **Depannage (Dépanneuse)** : Logique complexe de remorquage (positionnement, attachement, évacuation).
  - **TrafficLight & Road** : Classes utilitaires pour l'environnement statique et la signalisation.

# Diagramme de classes du simulateur de Smart City montrant la structure orientée objet, les relations d'héritage entre les véhicules et le rôle central de la classe Simulation.



## 2.2 Encapsulation et visibilité

Dans ce sous-projet, l'encapsulation est appliquée afin de protéger les données des objets et de garantir une manipulation sécurisée des véhicules.

Les attributs des classes sont déclarés en private ou protected pour empêcher un accès direct depuis l'extérieur.

Les getters et setters sont utilisés pour accéder ou modifier les attributs des véhicules, tels que la position, la vitesse, l'état ou la couleur. Cela permet de contrôler les modifications et d'assurer l'intégrité des données.

## Ces getters et setters illustrent l'encapsulation, en permettant de contrôler l'accès aux attributs des véhicules

```
// ===== Exemple de Getters et Setters =====
class Vehicle {
protected:
    float x, y;          // Position
    float speed;         // Vitesse
    bool moving;         // Déplacement

public:
    // Getters
    float GetX() const { return x; }
    float GetY() const { return y; }
    float GetSpeed() const { return speed; }
    bool IsMoving() const { return moving; }

    // Setters
    void SetX(float newX) { x = newX; }
    void SetY(float newY) { y = newY; }
    void SetSpeed(float s) { speed = s; }
    void SetMoving(bool state) { moving = state; }
};
```

## 2.3 Polymorphisme et fonctions virtuelles

La classe de base Vehicle définit des méthodes virtuelles Update et Draw pour le déplacement et le rendu des véhicules. Les classes dérivées (Ambulance, Depannage) redéfinissent ces méthodes afin de gérer des comportements spécifiques comme l'intervention d'urgence ou le remorquage.

Dans notre simulation, la classe Vehicle définit les comportements généraux des véhicules via des méthodes virtuelles :

```
class Vehicle {  
    public:  
        virtual void Update(bool stopForRed = false)  
        {  
            if (moving && !stopForRed) x += speed;  
            // Logique générale de déplacement  
        }  
  
        virtual void Draw() const {  
            DrawRectangle(x, y, VEHICLE_WIDTH,  
                          VEHICLE_HEIGHT, color);  
        }  
};
```

**Ensuite, les classes enfants redéfinissent ces méthodes pour des comportements spécifiques :**

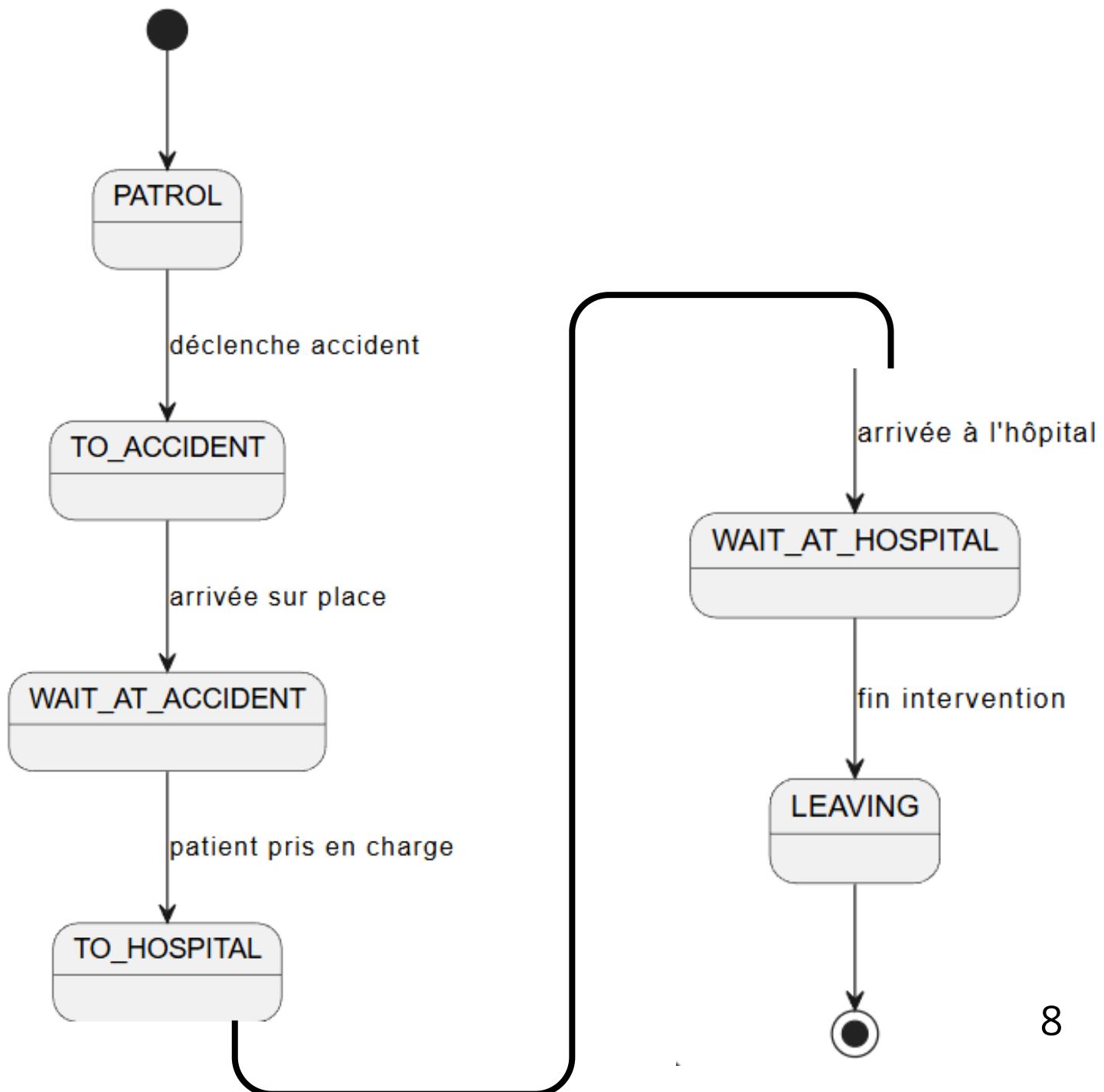
```
class Ambulance : public Vehicle {  
    public:  
        void Update(bool stopForRed = false) override {  
            // Ignore les feux rouges, suit un accident ou va à  
            // l'hôpital  
            if (state == TO_ACCIDENT) x += speed;  
            // Autres comportements spécifiques}  
            void Draw() const override {  
                DrawTexture(texture, x, y, WHITE); // Dessine  
                l'ambulance    }  
        };  
  
class Depannage : public Vehicle {  
    public:  
        void Update(bool stopForRed = false) override {  
            // Se déplace vers le véhicule à remorquer, logique de  
            // remorquage  
            if (!hasPickedUp) x -= speed;  
        }  
};
```

### 3. Design Patterns

#### 3.1 Pattern State

Le pattern State est utilisé pour gérer les différents comportements de l'ambulance : patrouille, déplacement vers l'accident, attente sur le lieu de l'accident, transport à l'hôpital et départ. Chaque état correspond à un comportement spécifique, ce qui facilite la lecture et la maintenance du code.

diagramme d'état UML de l'ambulance



## 3.2 Pattern Strategy

Le pattern *Strategy* est utilisé pour gérer les algorithmes de planification de trajet de manière flexible et interchangeable.

Dans notre projet de Smart City, il permet de changer dynamiquement l'algorithme de calcul de chemin utilisé par les véhicules prioritaires (ambulance, dépanneuse), selon le contexte.

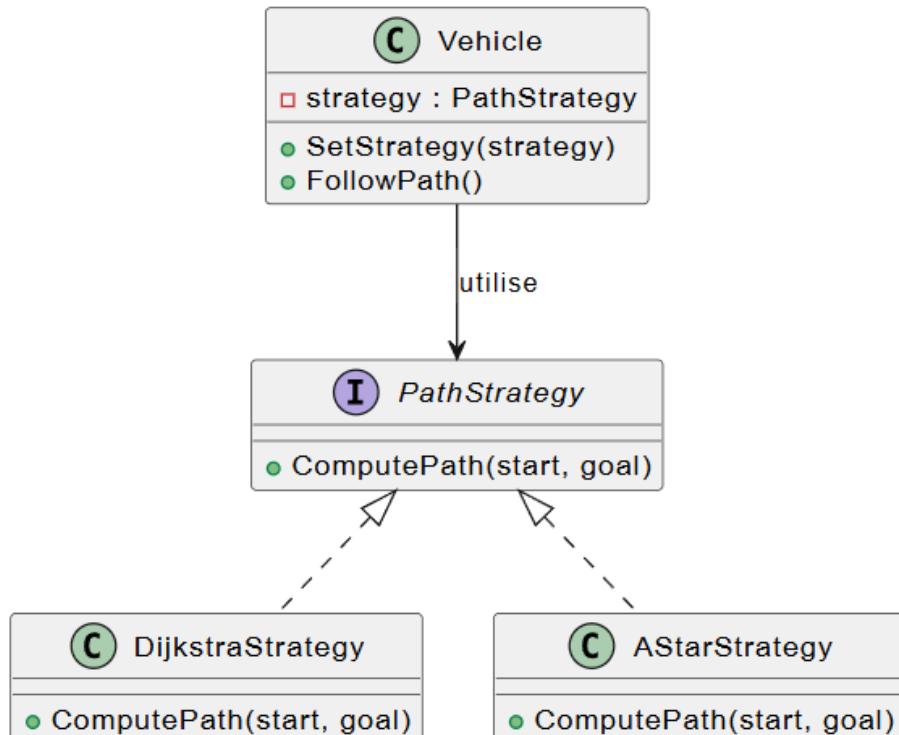
Les algorithmes implémentés sont :

- **Dijkstra** : garantit le chemin le plus court, mais avec un coût de calcul plus élevé.
- **A\*** : plus optimisé grâce à une heuristique, permettant un calcul plus rapide du trajet.

Structure conceptuelle

- PathStrategy : interface abstraite définissant le calcul du chemin.
- DijkstraStrategy et AStarStrategy : implémentations concrètes des algorithmes.
- Vehicle / Ambulance : utilise une stratégie de calcul de trajet sans connaître l'algorithme exact.

## UML Strategy



- Dijkstra est utilisé pour la précision
- A\* est utilisé pour l'optimisation et la rapidité

## Code pour Pattern Strategy (planification de trajet)

```

        // Strategy
class PathStrategy {
    public:
    virtual void ComputePath() = 0;
};

        // Dijkstra
class DijkstraStrategy : public PathStrategy {
    public:
    void ComputePath() override {
        // Calcul du plus court chemin
    }
};

        // A*
class AStarStrategy : public PathStrategy {
    public:
    void ComputePath() override {
        // Calcul optimisé avec heuristique
    }
};

        // Contexte
class Vehicle {
    PathStrategy* strategy;
    public:
    void SetStrategy(PathStrategy* s) { strategy = s; }
    void Navigate() { strategy->ComputePath(); }
};

```

## Utilisation des patterns State et Strategy

Notre projet implémente deux design patterns principaux : le pattern State pour la gestion des comportements de l'ambulance, et le pattern Strategy pour les algorithmes de planification de trajet (Dijkstra et A\*)

# 4. Implémentation des fonctionnalités

## 4.1 Gestion de la simulation

- **Utilisation des conteneurs STL (`std::vector`) pour véhicules et missions**

Afin de gérer dynamiquement les entités de la simulation, les conteneurs de la bibliothèque standard C++ (STL) ont été utilisés, en particulier `std::vector`.

Les véhicules sont stockés dans deux conteneurs distincts :

```
std::vector<std::unique_ptr<Vehicle>> vehiclesTop;
std::vector<std::unique_ptr<Vehicle>> vehiclesBottom;
```

Cette séparation permet de distinguer les flux de circulation des deux routes et de simplifier la gestion logique du trafic.

L'utilisation de `std::vector` offre une gestion flexible de la mémoire, permettant l'ajout et la suppression dynamique des véhicules pendant l'exécution de la simulation (apparition, disparition hors écran, accidents).

De plus, l'usage de `std::unique_ptr` garantit une gestion sécurisée des ressources et évite les fuites mémoire.

- **la boucle de simulation (Update / Draw)**

La simulation repose sur une boucle principale temps réel implémentée dans la fonction main.

Cette boucle est exécutée tant que la fenêtre de l'application est ouverte, via la condition `while (!WindowShouldClose())`.

```
while (!WindowShouldClose()) {
    float delta = GetFrameTime();
    sim.Update(delta);

    BeginDrawing();
    ClearBackground(SKYBLUE);
    sim.Draw();
    EndDrawing();
}
```

Elle est divisée en deux phases principales :

- **Phase Update :**

La méthode `sim.Update(delta)` est appelée à chaque itération afin de mettre à jour l'état logique du système.

Cette phase gère le déplacement des véhicules, le fonctionnement des feux de circulation, la détection des accidents ainsi que le déclenchement des interventions.

- **Phase Draw :**

La méthode `sim.Draw()` assure le rendu graphique de la simulation à l'aide de la bibliothèque Raylib.

Elle permet l'affichage des routes, des véhicules, des signalisations et des alertes visuelles.

Cette séparation entre logique et affichage garantit une simulation fluide et facilement maintenable.

## 4.2 Spawn des véhicules

Le mécanisme de génération des véhicules (Spawn) permet de simuler un flux de circulation dynamique et variable.

Les véhicules ne sont pas créés au démarrage de l'application, mais apparaissent progressivement au cours de la simulation afin de reproduire des conditions de trafic réalistes.

- **Génération aléatoire : position, vitesse, couleur**

- 1. Génération aléatoire de la position (voie)

```
int lane = GetRandomValue(0, 2);
```

Cela permet :

- répartir les véhicules sur trois voies,
- d'éviter un trafic uniforme.

- 2. Génération aléatoire de la vitesse

```
float speed = 2.0f + GetRandomValue(0, 5) / 10.0f;
```

- La vitesse varie légèrement pour chaque véhicule
- Cela crée des différences de comportement (véhicules lents/rapides)
- Améliore le réalisme de la simulation

### 3. Génération aléatoire de la couleur

```
Color c = {  
    (unsigned char)GetRandomValue(80, 255),  
    (unsigned char)GetRandomValue(80, 255),  
    (unsigned char)GetRandomValue(80, 255),  
    255  
};
```

-Chaque véhicule possède une couleur distincte

-Facilite la visualisation et la différenciation graphique

Les propriétés des véhicules sont générées de manière aléatoire afin d'introduire de la variabilité dans la simulation.

La voie de circulation, la vitesse et la couleur sont choisies dynamiquement à chaque création de véhicule.

Cette approche permet de reproduire des comportements hétérogènes et d'éviter une circulation uniforme.

- **SpawnCarTop/Bottom**

**SpawnCarTop()** : génère un véhicule sur la route supérieure (gauche → droite)

```
void SpawnCarTop() {  
    int lane = GetRandomValue(0, 2);  
    float speed = 2.0f + GetRandomValue(0, 5) / 10.0f;  
    Color c = { (unsigned char)GetRandomValue(80, 255), (unsigned  
    char)GetRandomValue(80, 255), (unsigned  
    char)GetRandomValue(80, 255), 255 };  
    vehiclesTop.push_back(std::make_unique<Car>(-200,  
laneYTop[lane], speed, c, true, carImages[GetRandomValue(0, 4)]));  
}
```

**SpawnCarBottom()** : génère un véhicule sur la route inférieure (droite → gauche)

```
void SpawnCarBottom() {  
    int lane = GetRandomValue(0, 2);  
    float speed = 2.0f + GetRandomValue(0, 5) /  
10.0f;  
    Color c = { ... };  
    vehiclesBottom.push_back(  
        std::make_unique<Car>(SCREEN_WIDTH +  
200, laneYBottom[lane], speed, c, false,  
carImages[GetRandomValue(0, 4)])  
    );  
}
```

- **le déclenchement du Spawn (timer)**

La génération des véhicules est déclenchée par un temporisateur basé sur le temps réel (delta time).

Cette approche permet de contrôler la fréquence d'apparition des véhicules et de simuler différents niveaux de densité de trafic.

```
carSpawnTimerTop += delta;  
if (carSpawnTimerTop >= GetRandomValue(40, 70) /  
    10.0f) {  
    carSpawnTimerTop = 0.0f;  
    SpawnCarTop(); }
```

## 4.3 Interaction véhicules / feux

Dans la simulation, chaque véhicule adapte son comportement en fonction des feux de circulation et des véhicules prioritaires.

### 1. Gestion des feux

Les feux (TrafficLight) alternent entre rouge et vert selon un cycle prédéfini. Les véhicules classiques s'arrêtent lorsqu'ils rencontrent un feu rouge :

```
// Exemple pour un véhicule sur la voie du bas  
float stopX = lightBottom.GetStopLineX(true);  
if (lightBottom.IsRed() && fabs(v->GetX() - stopX) < 50)  
    stop = true;  
v->SetForcedStop(stop);  
v->Update(stop);
```

-**GetStopLineX()** calcule la position de la ligne d'arrêt.

-**IsRed()** vérifie l'état du feu.

-Si le véhicule est proche et que le feu est rouge, il s'arrête (forcedStop).

### 2. Couloir vert pour véhicules prioritaires

Les véhicules prioritaires (ambulances et dépanneuses) ne respectent pas les feux et peuvent créer un couloir pour se déplacer rapidement :

```
if (v->isReckless) { // véhicule prioritaire  
    stopForRed = false; // ignore les feux  
    forcedStop = false;  
}
```

De plus, les véhicules normaux peuvent changer de voie automatiquement pour laisser passer l'ambulance ou le dépanneur:

```
if (emergencyVehicle && emergencyVehicle->IsMoving()) {  
    if (fabs(v->GetTargetY() - emergencyVehicle->GetTargetY()) < 5.0f) {  
        float dist = emergencyVehicle->GetX() - v->GetX();  
        if (dist > 0 && dist < 350.0f) {  
            // changement de voie pour dégager le passage  
            int targetLane = (currentLaneIdx + 1) % 3;  
            v->SetTargetY(laneYBottom[targetLane]);  
            v->SetChangedLane(true); } } }
```

-Les véhicules normaux détectent la présence d'un véhicule prioritaire proche.

-Ils changent de voie si possible pour créer un couloir vert.

-Cette logique permet d'éviter les blocages et de simuler un comportement réaliste dans les intersections.

## 4.4 Gestion des accidents

La simulation gère automatiquement les accidents entre véhicules. Les étapes principales sont: détection, passage en état actif, et intervention des véhicules prioritaires.

### 1. Détection automatique

Lorsqu'un véhicule se rapproche trop d'un autre dans la même voie, un accident peut être déclenché :

```
if (fabs(v1->GetTargetY() - v2->GetTargetY()) < 5.0f)  
{  
    float dist = v1->GetX() - v2->GetX();  
    if (dist < 400 && dist > 110) {  
        currentAccident.pending = true;  
        currentAccident.car1 = v2;  
        currentAccident.car2 = v1;  
        v1->isReckless = true;  
        v2->isAccidentTarget = true;  
        v1->laneLock = true;  
        v2->laneLock = true;  
        v1->SetSpeed(v1->GetSpeed() * 2.8f);  
        v2->SetSpeed(v2->GetSpeed() * 0.4f);  
    } }
```

- Les véhicules impliqués sont identifiés (car1 et car2).
- L'accident est initialement en état pending (impact imminent).
- Les véhicules peuvent ajuster leur vitesse ou leur comportement avant la collision.

## 2. Passage à l'état actif

Quand les véhicules se touchent réellement, l'accident devient actif :

```
if (dist < VEHICLE_WIDTH - 10.0f && dist > -VEHICLE_WIDTH) {
    currentAccident.pending = false;
    currentAccident.active = true;
    currentAccident.car1->isCrashed = true;
    currentAccident.car2->isCrashed = true;
    currentAccident.car2->isReckless = false;
    currentAccident.car1->SetMoving(false);
    currentAccident.car2->SetMoving(false);
    currentAccident.x = currentAccident.car1->GetX() +
    VEHICLE_WIDTH/2;
    currentAccident.y = currentAccident.car1->GetY(); }
```

- Les véhicules sont arrêtés (SetMoving(false)).
- Les drapeaux isCrashed indiquent qu'ils sont impliqués dans l'accident.
- Les positions x et y de l'accident sont mémorisées pour l'intervention.

## 3. Assignation ambulance et dépanneur

Une fois l'accident actif :

- **Ambulance:** se dirige vers la position de l'accident pour "prendre en charge" les victimes.

```
for (auto& v : vehiclesBottom) {
    if (v->IsAmbulance()) {
        static_cast<Ambulance*>(v.get())-
        >AssignAccident(currentAccident.x,
        currentAccident.y);
        v->SetTargetY(currentAccident.y);
    } }
```

- **Dépanneuse:** se déplace jusqu'à l'accident pour remorquer les véhicules impliqués.

```

if (activeTow && activeTow->hasPickedUp &&
currentAccident.active) {
    if (currentAccident.car1) {
        currentAccident.car1->isTowed = true;
        currentAccident.car1->isCrashed = false;
        currentAccident.car1->towOffsetX = 100.0f;
        currentAccident.car1->SetY(activeTow->GetY());
    }
    if (currentAccident.car2) {
        currentAccident.car2->isTowed = true;
        currentAccident.car2->isCrashed = false;
        currentAccident.car2->towOffsetX = 200.0f;
        currentAccident.car2->SetY(activeTow->GetY());
    }
    currentAccident.active = false;
}

```

-Les voitures accidentées suivent la dépanneuse après leur récupération (isTowed).

-La gestion de l'accident est terminée après le remorquage.

## 4.5 Planificateur de trajets

Le planificateur de trajets permet à chaque véhicule de s'adapter dynamiquement à son environnement :

### 1. Éviter les obstacles :

- Accidents (currentAccident.active)
- Véhicules ralentis ou arrêtés
- Véhicules prioritaires (ambulances, dépanneuses)

### 1. Gestion des embouteillages :

- Les véhicules respectent une distance minimale de sécurité (SAFE\_DISTANCE) entre eux.
  - Ils peuvent s'arrêter si la voie est bloquée ou s'ils approchent d'un feu rouge.

### **3.Création de couloirs verts pour véhicules prioritaires :**

- Les véhicules normaux changent de voie si un véhicule prioritaire approche.
- Les ambulances et dépanneuses ignorent les feux et les obstacles mineurs.

### **4.pseudo-code du planificateur**

Pour chaque véhicule v :

Si v est prioritaire (ambulance/dépanneuse) :

Ignorer feux rouges

Suivre trajet direct vers destination (accident ou hôpital)

Sinon :

Vérifier obstacles devant :

Si distance < SAFE\_DISTANCE :

Stopper ou ralentir

Vérifier accident actif :

Si accident sur même voie et proche :

Changer de voie si possible

Vérifier véhicule prioritaire :

Si proche et sur même voie :

Changer de voie pour dégager passage

Vérifier feu rouge :

Si proche et rouge :

Stopper

Mettre à jour position en fonction de la vitesse et de la direction

### **4.6 Gestion des exceptions**

Dans une simulation complexe, des situations imprévues peuvent se produire :

- **Destination inatteignable** : un véhicule prioritaire (ambulance ou dépanneuse) peut être bloqué par des obstacles ou un accident non remorqué.
- **Saturation du trafic** : trop de véhicules sur une même voie entraînent des blocages ou ralentissements extrêmes.

Pour gérer ces cas, la simulation utilise des blocs try/catch afin d'éviter les plantages et de notifier l'erreur.

- gestion d'erreurs en pseudo-code / C++

```

try {
    // Tentative de mise à jour de la position du véhicule
    v->Update(stopForRed);

    // Vérification de la destination
    if (v->GetX() > SCREEN_WIDTH + 500 || v->GetX() < -500) {
        throw std::runtime_error("Destination inatteignable !");
    }

    // Vérification de saturation
    if (vehiclesBottom.size() > MAX_VEHICLES) {
        throw std::overflow_error("Saturation de la voie détectée
        !");
    }
}

catch (const std::overflow_error& e) {
    std::cout << "Erreur de trafic : " << e.what() << std::endl;
    // Réduire le spawn de véhicules pour désengorger
    carSpawnTimerBottom += 1.0f;
}

catch (const std::runtime_error& e) {
    std::cout << "Erreur véhicule : " << e.what() << std::endl;
    // Retirer le véhicule problématique
    v->toBeRemoved = true;
}

catch (...) {
    std::cout << "Erreur inconnue détectée !" << std::endl;
}

```

- **try**: le code susceptible de générer des exceptions.
- **catch**: capture des exceptions spécifiques (overflow\_error, runtime\_error) et applique une stratégie corrective.
- **catch(...)**: capture toute exception inconnue pour éviter le crash de la simulation.

## 4.7 Logiciel temps réel

La simulation est conçue comme un système temps réel, où chaque véhicule met à jour sa position, sa vitesse et son comportement à chaque cycle de rendu. Cela permet de gérer :

- Le mouvement continu des véhicules.
  - Les réactions aux obstacles, accidents et feux de circulation.
  - L'adaptation à la présence de véhicules prioritaires.
- 
- **mise à jour des véhicules (C++)**

```
for (auto& v : vehiclesBottom) {  
    bool stop = false;  
  
    // Vérification feu rouge  
    float stopX = lightBottom.GetStopLineX(true);  
    if (lightBottom.IsRed() && fabs(v->GetX() - stopX) < 50)  
        stop = true;  
  
    // Vérification collision avec d'autres véhicules  
    for (auto& other : vehiclesBottom) {  
        if (v.get() == other.get()) continue;  
        if (fabs(v->GetTargetY() - other->GetTargetY()) < 5.0f) {  
            float dist = other->GetX() - v->GetX();  
            if (dist > 0 && dist < SAFE_DISTANCE) {  
                stop = true;  
                break;  
            }  
        }  
    }  
  
    v->SetForcedStop(stop);    // Appliquer l'arrêt si nécessaire  
    v->Update(stop);          // Mise à jour de la position et vitesse  
}
```

- **SetForcedStop(stop):** indique au véhicule s'il doit s'arrêter.
- **Update(stop):** calcule la nouvelle position en fonction de la vitesse, de la direction et de l'état d'arrêt.
- Les véhicules adaptent leur trajectoire et vitesse à chaque frame, ce qui garantit un rendu temps réel fluide.

## 5. Visualisation avec Raylib

### 5.1 Représentation graphique

La simulation utilise Raylib, une bibliothèque C++ légère pour la création graphique en 2D , afin de représenter visuellement les éléments de la circulation.

#### 1. Routes et voies

- Les routes sont dessinées en rectangles gris foncé pour simuler l'asphalte.
- Les voies sont matérialisées par des lignes blanches ou des bandes jaunes pour les séparations.
- Exemple pour la route supérieure:

```
DrawRectangle(0, ROAD_Y_TOP, SCREEN_WIDTH,
ROAD_HEIGHT, {40, 40, 40, 255});
for (int i = 1; i < 3; i++) {
    DrawLine(0, ROAD_Y_TOP + i * LANE_HEIGHT,
SCREEN_WIDTH, ROAD_Y_TOP + i * LANE_HEIGHT,
Fade(WHITE, 0.7f));
}
for (int i = 0; i < SCREEN_WIDTH; i += 80) {
    DrawRectangle(i, ROAD_Y_TOP + (ROAD_HEIGHT
/ 2) - 3, 40, 6, YELLOW);
}
```

#### 2. Véhicules

- Chaque type de véhicule est représenté par une texture ou une couleur distincte :
  - Voitures classiques → textures variées (car.png, car2.png, etc.).
  - Ambulances → texture spécifique ambulance.png.
  - Dépanneuses → texture depannage.png.

- Les véhicules changent de couleur ou de texture lorsqu'ils sont impliqués dans un accident:

```
Color drawColor = WHITE;
if (isCrashed) drawColor = RED;
DrawTexturePro(texture, source, dest, origin,
rotation, drawColor);
```

### 3. Bâtiments et décor

- Hôpital: texture placée à côté de la route pour la visibilité des ambulances.
- Maisons: décor pour rendre la simulation plus réaliste et contextualisée.

```
DrawTexture(hospitalTexture, 10,
ROAD_Y_BOTTOM + ROAD_HEIGHT + 10, WHITE);
DrawTexture(houseTextures[0], 250, 410, WHITE);
DrawTexture(houseTextures[1], 580, 440, WHITE);
DrawTexture(houseTextures[2], 700, 400, WHITE);
```

### 4. Indications visuelles supplémentaires

- Alertes pour véhicules prioritaires: bandes rouges clignotantes sur les bords de l'écran si une ambulance est active.
- Messages texte pour guider l'utilisateur:

```
DrawText("Press 'E' for Ambulance", 10, 10, 20, WHITE);
DrawText("SAFE LANE: Bottom Lane (3)", 10, 85, 20, GREEN);
```



## 5.2 Interface utilisateur

L'interface utilisateur de la simulation a été conçue pour offrir une lecture claire de l'état du trafic, des missions en cours et des situations d'urgence. Elle repose sur les fonctionnalités graphiques temps réel de Raylib.

### 1. Affichage des missions

Des messages textuels sont affichés en permanence à l'écran afin d'informer l'utilisateur des actions possibles et des missions actives :

```
DrawText("Press 'E' for Ambulance", 10, 10, 20, WHITE);
DrawText("Press 'D' for Tow Truck", 10, 35, 20, WHITE);
DrawText("Press 'A' for Accident", 10, 60, 20, WHITE);
```

- L'utilisateur peut déclencher manuellement une ambulance, une dépanneuse .
- Ces indications facilitent l'interaction avec la simulation sans menu complexe.

### 2. Alertes visuelles

Lorsqu'une ambulance est active, une alerte visuelle clignotante apparaît sur les bords de l'écran pour signaler une situation d'urgence :

```
if (screenAlertOn) {
    DrawRectangle(0, 0, 20, SCREEN_HEIGHT, Fade(RED, 0.7f));
    DrawRectangle(SCREEN_WIDTH - 20, 0, 20, SCREEN_HEIGHT,
    Fade(RED, 0.7f));
}
```

- L'effet de clignotement est synchronisé avec le temps (screenAlertTimer).
- Cela attire immédiatement l'attention de l'utilisateur sur l'intervention en cours.

### 3. Sirène sonore

Une sirène audio est jouée lors de l'appel d'une ambulance :

```
siren = LoadSound("siren.wav");
PlaySound(siren);
```

- Le son renforce l'immersion et la perception des situations critiques.
- Il est directement lié aux événements de la simulation (accident, intervention).

### 4. Messages contextuels

Des messages dynamiques informent sur l'état du trafic :

```
if(currentAccident.active)
    DrawText("ACCIDENT ACTIVE!", SCREEN_WIDTH/2 - 100, 50,
20, RED);

if(currentAccident.pending)
    DrawText("IMPACT IMMINENT...", SCREEN_WIDTH/2 - 110,
50, 20, ORANGE);
```

- **ACCIDENT ACTIVE** : collision confirmée.
- **IMPACT IMMINENT** : accident en cours de déclenchement (pending).

## 6.1 Tests d'héritage et de polymorphism

*Objectif*

Vérifier que :

- L'héritage entre les classes est correctement implémenté
  - Les méthodes virtual sont bien redéfinies
  - Les getters et setters fonctionnent correctement
- Classes concernées (exemple)
- Voiture, Ambulance, Depanneur (classes filles)
  - Vehicule (classe mère)

### TEST 1 : VÉRIFICATION DE L'HÉRITAGE

```
#include <cassert>
#include "Vehicule.h"
#include "Voiture.h"

void testHeritage() {
    Vehicule* v = new Voiture(100, 200);

    // Test polymorphisme
    assert(v->getType() == "Voiture");

    delete v;
}
```

## TEST 2 : VÉRIFICATION DES GETTERS ET SETTERS

```
void testGettersSetters() {  
    Voiture v(50, 60);  
  
    v.setVitesse(80);  
    assert(v.getVitesse() == 80);  
  
    v.setPosition(100, 120);  
    assert(v.getX() == 100);  
    assert(v.getY() == 120);  
}
```



## 6.2 Tests déplacements et priorités

*Objectif*

Vérifier :

- Le déplacement correct des véhicules
- Le respect des priorités (feux tricolores, ambulance)
- L'évitement des collisions

### TEST 3 : DÉPLACEMENT D'UN VÉHICULE

```
void testDeplacement() {  
    Voiture v(0, 0);  
    v.setVitesse(10);  
  
    v.update(); // déplacement  
    assert(v.getX() == 10);  
}
```

✓ Résultat attendu :

Le véhicule avance selon sa vitesse.

## TEST 4 : PRIORITÉ AMBULANCE

```
void testPrioriteAmbulance() {  
    Ambulance a(0, 0);  
    Voiture v(10, 0);  
  
    bool priorite = a.aLaPrioriteSur(v);  
    assert(priorite == true);  
}
```

✓ Résultat attendu :

L'ambulance est toujours prioritaire.

## TEST 5 : COLLISION AVOIDANCE

```
void testCollisionAvoidance() {  
    Voiture v1(50, 50);  
    Voiture v2(50, 50);  
  
    bool collision = v1.detectCollision(v2);  
    assert(collision == true);  
}
```

✓ Résultat attendu :

La collision est détectée et le mouvement est  
bloqué ou ajusté.

## *6.3 Tests accidents et planification*

Objectif

Vérifier :

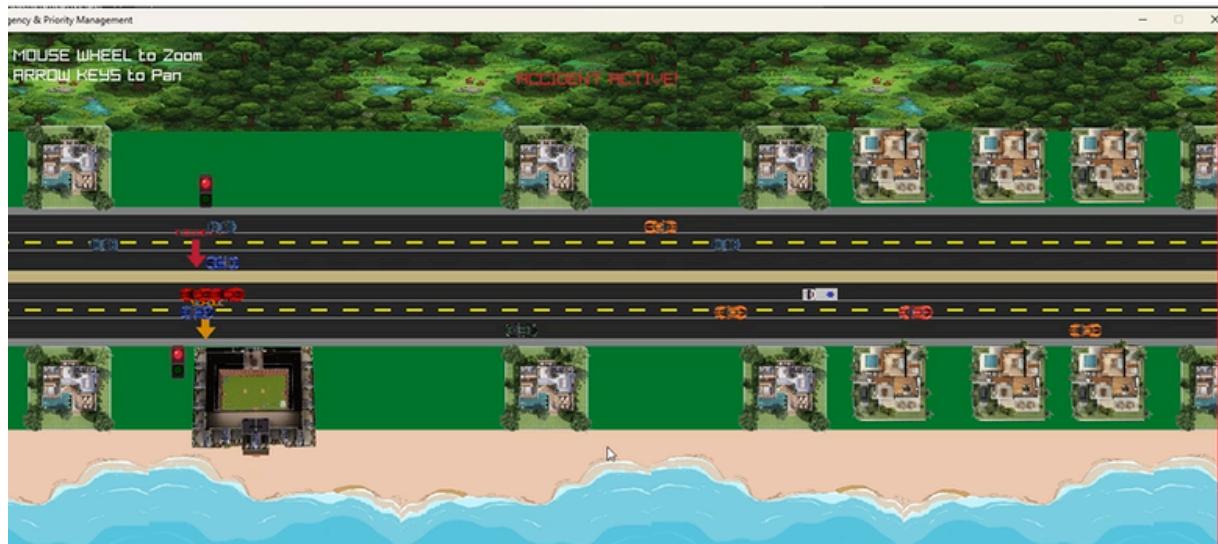
- Le déclenchement d'un accident
- L'intervention automatique de l'ambulance ou du dépanneur
- La planification correcte des actions

### **TEST 6 : DÉCLENCHEMENT D'UN ACCIDENT**

```
void testAccident() {  
    Voiture v1(100, 100);  
    Voiture v2(100, 100);  
  
    v1.checkAccident(v2);  
    assert(v1.estEnAccident() == true);  
}
```

### **TEST 7 : INTERVENTION AMBULANCE**

```
void testInterventionAmbulance() {  
    Ambulance a(0, 0);  
    bool intervention = a.intervenirAccident(100, 100);  
  
    assert(intervention == true);  
}
```



## 7.1 Étapes de la démonstration

### ÉTAPE 1 : LANCEMENT DE LA SIMULATION

Au démarrage de l'application :

- La fenêtre graphique est initialisée avec raylib
- Les routes supérieure et inférieure sont dessinées
- Les feux de circulation sont initialisés en état rouge/vert
- Les véhicules normaux (voitures) apparaissent progressivement sur les deux routes

### ÉTAPE 2 : CIRCULATION NORMALE DES VÉHICULES

Dans une situation normale :

- Les voitures se déplacent dans leurs voies respectives
- Elles respectent les feux tricolores
- Elles maintiennent une distance de sécurité (SAFE\_DISTANCE)
- Les collisions sont évitées automatiquement



### ÉTAPE 3 : APPARITION D'UNE AMBULANCE (SITUATION D'URGENCE)

Après un certain temps :

- Une ambulance est générée automatiquement
- Elle se déplace plus rapidement que les véhicules normaux
- Elle ignore les feux rouges
- Elle bénéficie d'une priorité absolue

Dans le code :

Vehicle::Update(false);



## ÉTAPE 4 : PRIORITÉ DE L'AMBULANCE ET CHANGEMENT DE VOIE

Lorsque l'ambulance s'approche :

- Les voitures situées sur la voie centrale détectent sa proximité
- Elles changent automatiquement de voie (gauche ou droite)
- Cela libère la voie centrale pour permettre le passage de l'ambulance

## ÉTAPE 5 : INTERACTION AVEC LES FEUX DE CIRCULATION

Pendant toute la durée de l'urgence :

- Les véhicules normaux s'arrêtent au feu rouge
- L'ambulance continue sa route sans interruption
- Le feu ne bloque jamais le véhicule prioritaire

📸 Capture d'écran : ambulance traversant un feu rouge



## ÉTAPE 6 : ARRIVÉE DE L'AMBULANCE À L'HÔPITAL

Sur la route inférieure : Sur la route inférieure :

- L'ambulance change de voie pour entrer dans la zone de l'hôpital
- Elle s'arrête pendant 5 secondes (temps de prise en charge)
- Elle reprend ensuite sa route

### CODE :

```
amb->atHospital = true;  
amb->hospitalTimer += GetFrameTime();
```

📸 *Capture d'écran : ambulance arrêtée à l'hôpital*



## ÉTAPE 7 : RETOUR PROGRESSIF À LA NORMALE

Après l'intervention :

- L'ambulance quitte la scène
- Les véhicules reprennent leur comportement normal
- Les feux régulent à nouveau le trafic
- La simulation revient à un état stable

 Capture d'écran : trafic normal après intervention



## *8.1 Synthèse du travail*

### Objectifs initiaux

L'objectif principal de ce projet était de concevoir et implémenter une simulation de trafic routier bidirectionnel en C++ en utilisant la bibliothèque raylib, intégrant :

- La circulation de véhicules normaux
- La gestion des feux tricolores
- La priorité des véhicules d'urgence
- Une logique réaliste d'interactions entre les entités

### **1. SIMULATION GRAPHIQUE EN TEMPS RÉEL**

- Fenêtre interactive gérée par raylib
- Rendu fluide à 60 FPS
- Affichage dynamique des routes, voies et véhicules

### **2. ARCHITECTURE ORIENTÉE OBJET**

- Classe Vehicle comme base commune
- Héritage via Car et Ambulance
- Utilisation du polymorphisme (virtual Update())

### **3. GESTION INTELLIGENTE DU TRAFIC**

- Respect des feux de circulation
- Maintien de la distance de sécurité (SAFE\_DISTANCE)
- Évitement automatique des collisions
- Changement dynamique de voie

### **4. GESTION DES SITUATIONS D'URGENCE**

- Apparition aléatoire des ambulances
- Priorité absolue sur les autres véhicules
- Passage aux feux rouges
- Arrêt temporaire à l'hôpital (planification)

## **5. SYSTÈME AUTONOME ET STABLE**

- Suppression automatique des véhicules hors écran
- Retour progressif à un état de circulation normal
- Aucun blocage du trafic observé

### *8.2 Perspectives d'évolution*

#### **1. INTÉGRATION AVEC UN MODULE « TRAFFIC CORE »**

Un module centralisé pourrait être ajouté pour :

- Collecter les données de trafic (densité, vitesse moyenne)
- Ajuster dynamiquement les feux tricolores
- Prioriser les axes en cas d'urgence

#### **2. INTÉGRATION D'UN MODULE « SMART PARKING »**

La simulation pourrait être étendue avec :

- Des parkings intelligents
- Indication des places libres
- Orientation automatique des véhicules
- Réduction des embouteillages urbains

### **3. GESTION AVANCÉE DES INCIDENTS (ACCIDENTS / DÉPANNEUR)**

Évolutions possibles :

- Détection explicite des accidents
- Blocage temporaire de voies
- Apparition d'un dépanneur
- Déviation dynamique du trafic

### **4. COMMUNICATION INTER-MODULES (V2X)**

Ajout de :

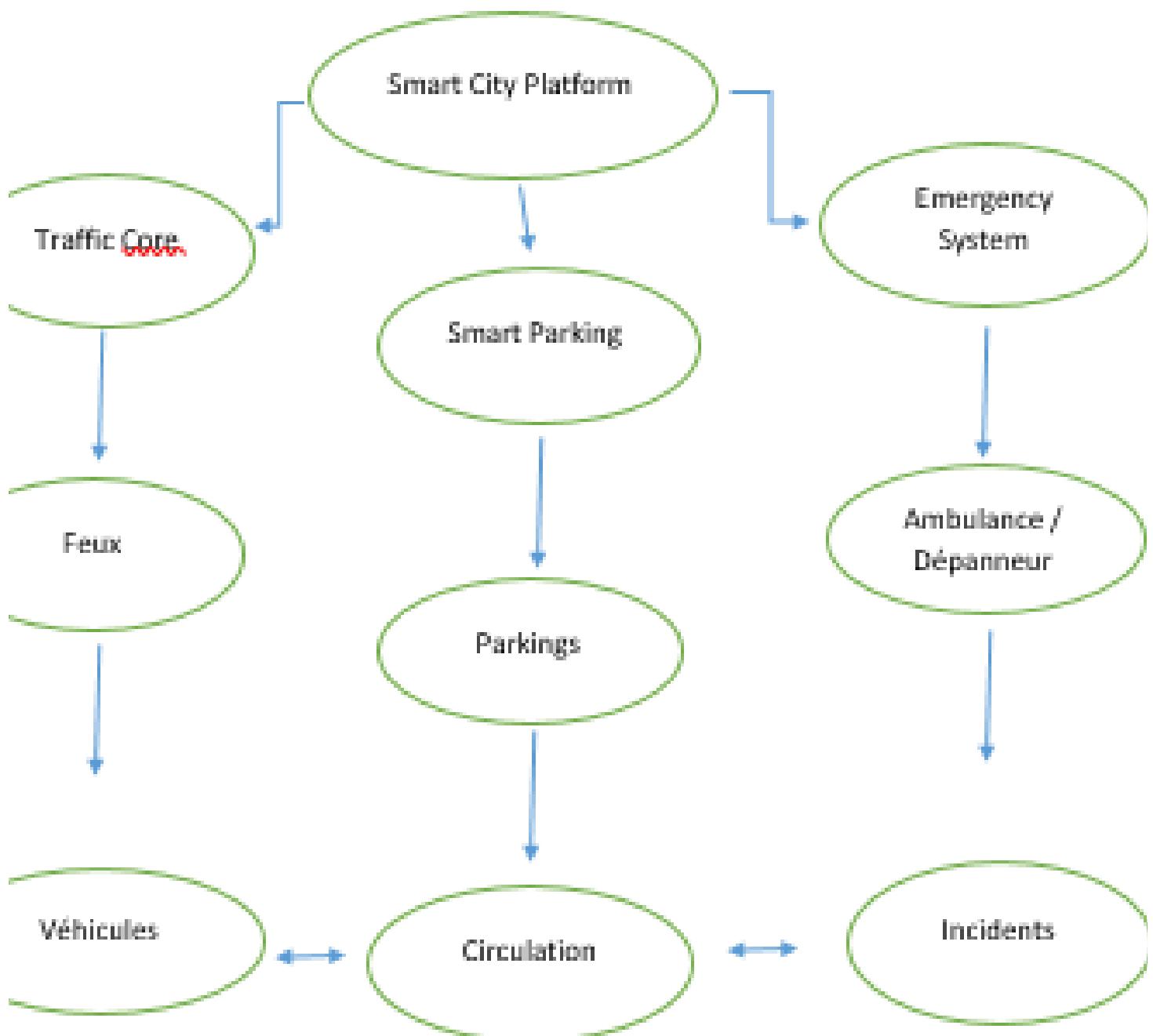
- Communication véhicule-véhicule (V2V)
- Communication véhicule-infrastructure (V2I)
- Partage en temps réel des informations de trafic

### **5. EXTENSION VERS UNE SMART CITY COMPLÈTE**

L'application pourrait devenir un composant d'une plateforme globale intégrant :

-  Traffic Management
-  Smart Parking
-  Emergency Services
-  Urban Simulation
-  Analyse de données en temps réel

## DIAGRAMME CONCEPTUEL SMART CITY)



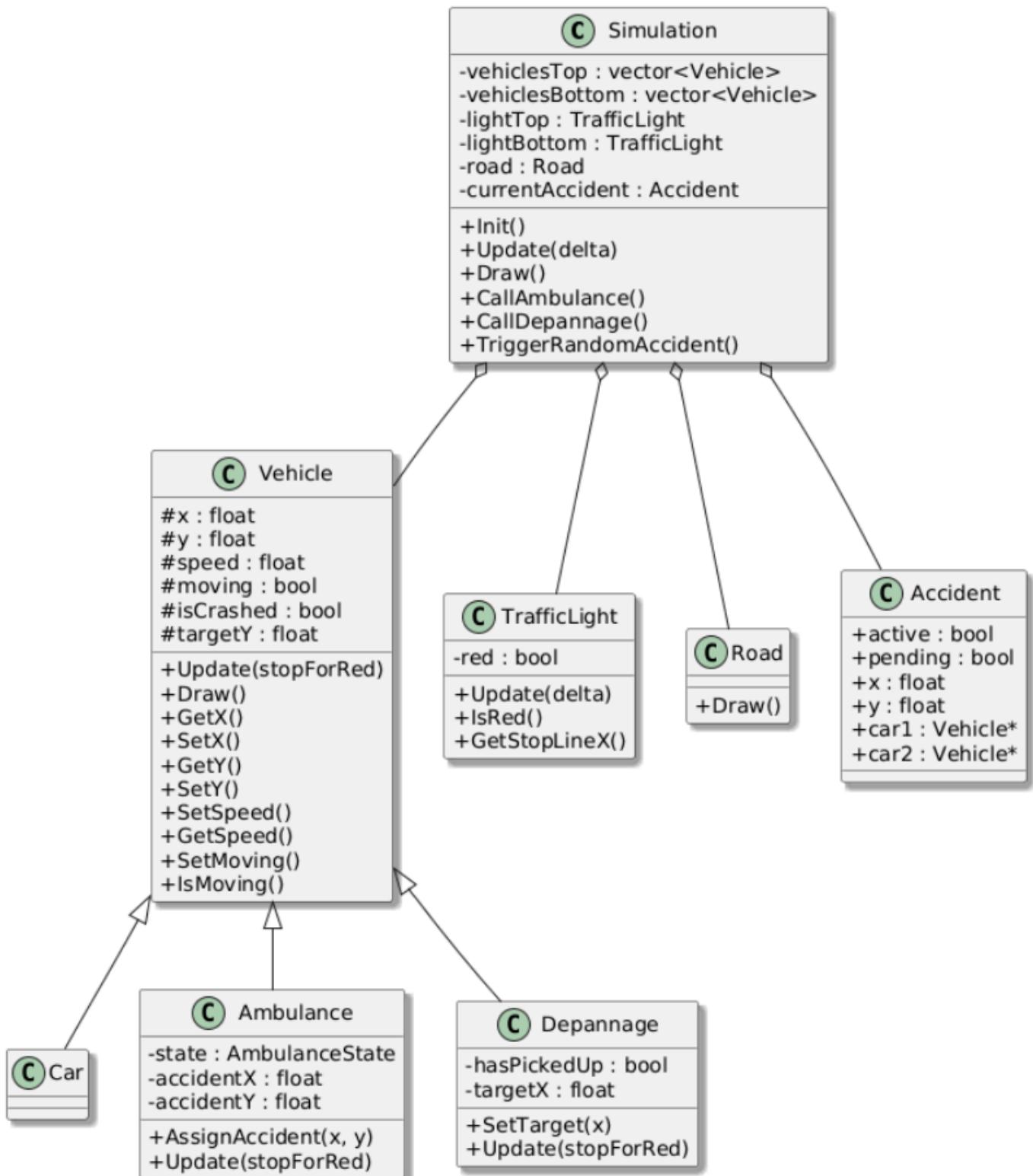
## Conclusion

Ce projet a permis de réaliser une simulation de trafic routier en utilisant le langage C++ et la bibliothèque raylib. Grâce à ce travail, nous avons appliqué les concepts de la programmation orientée objet tels que l'héritage et le polymorphisme.

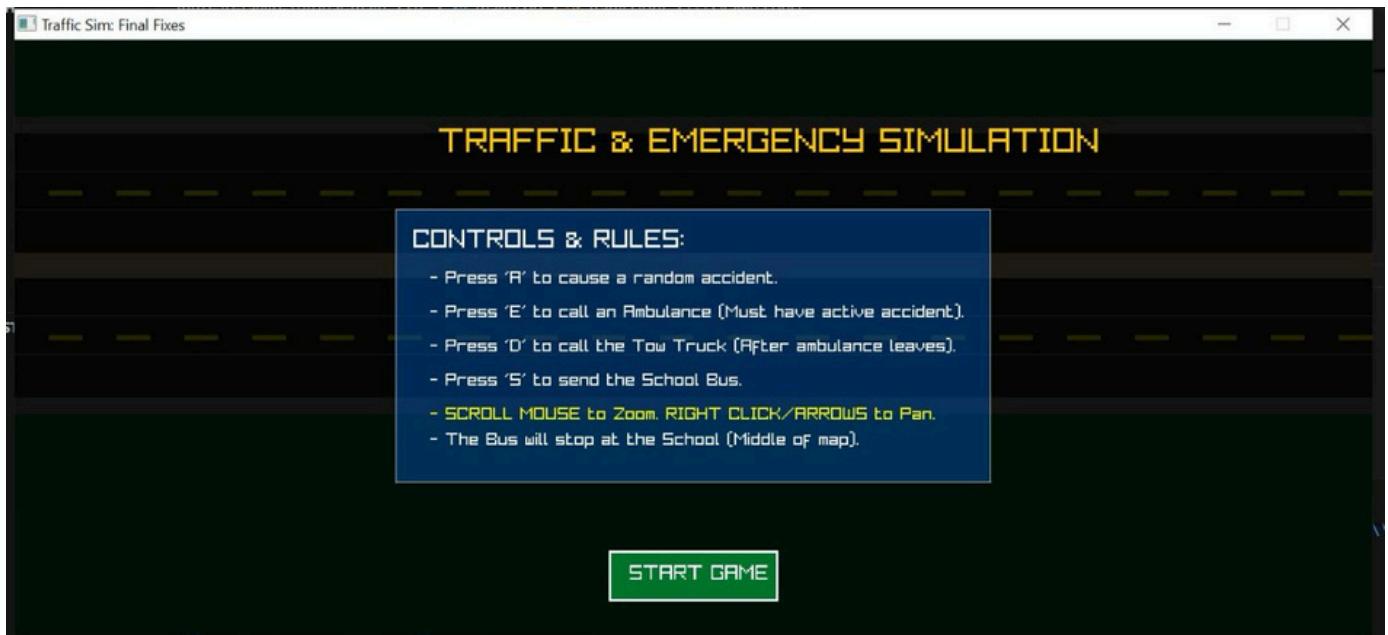
La simulation montre le déplacement des véhicules, le fonctionnement des feux de circulation et la gestion des priorités, notamment pour l'ambulance. Le comportement du système reste cohérent et stable pendant l'exécution. Ce projet constitue une bonne base pour comprendre la simulation du trafic et peut être amélioré et étendu par la suite vers des applications plus avancées, comme les systèmes de Smart City.

# Annexes

## Diagramme UML détaillé du projet Smart City, incluant les classes et leurs relations



# Interface Raylib



## Code source complet

Le projet est disponible sur GitHub :  
[https://github.com/Oboualiti/cpp\\_projet](https://github.com/Oboualiti/cpp_projet)

## Vidéo de démonstration.

[lien de la video](#)