Rapport du projet tuteuré d'AP2 :

Traffic Jam

Réalisé par SILVERIO Anthony et ROUDEAU Gaëtan

Table des matières

Introduction

G.E.R.T.R.U.D.E (Gestion Electronique de Régulation du Trafic Routier Urbain Défiant les Embouteillages)

« Ce système permet de fluidifier le trafic automobile, grâce à une perception en temps réel du nombre de voitures en certains points de la ville. Les feux de signalisation peuvent, toujours grâce à Gertrude, être adaptés quand les forces de police responsables de la circulation peuvent dès lors intervenir au plus près des événements.

Le bénéfice pour la CUB est double. La ville a effectivement gagné en fluidité de circulation, tout en conservant les particularités de son patrimoine d'exception. En outre, Gertrude participe à la renommée de Bordeaux sur son savoir-faire en matière de gestion urbaine. »

http://www.parolesdelus.com

I] Présentation générale

A. Principe

Le projet tuteuré d'AP2 s'intitule Traffic Jam. C'est un jeu de gestion de trafic routier qui met en scène un ensemble de véhicules de plusieurs types et de vitesses différentes (voiture : vitesse normale, police : vitesse rapide et camion : vitesse lente) qui circulent sur des routes. Le joueur doit ainsi atteindre un nombre précis de véhicules ayant traversés la carte afin de réussir le niveau et passer au suivant. Pour cela le joueur doit éviter les collisions entre véhicules aux intersections de routes grâce à des feux dont il peut modifier la couleur en cliquant dessus (vert : les véhicules passent et rouge ils s'arrêtent). Chaque véhicule passé avec succès lui rapporte un nombre de point dépendant de son type.

B. Objectifs

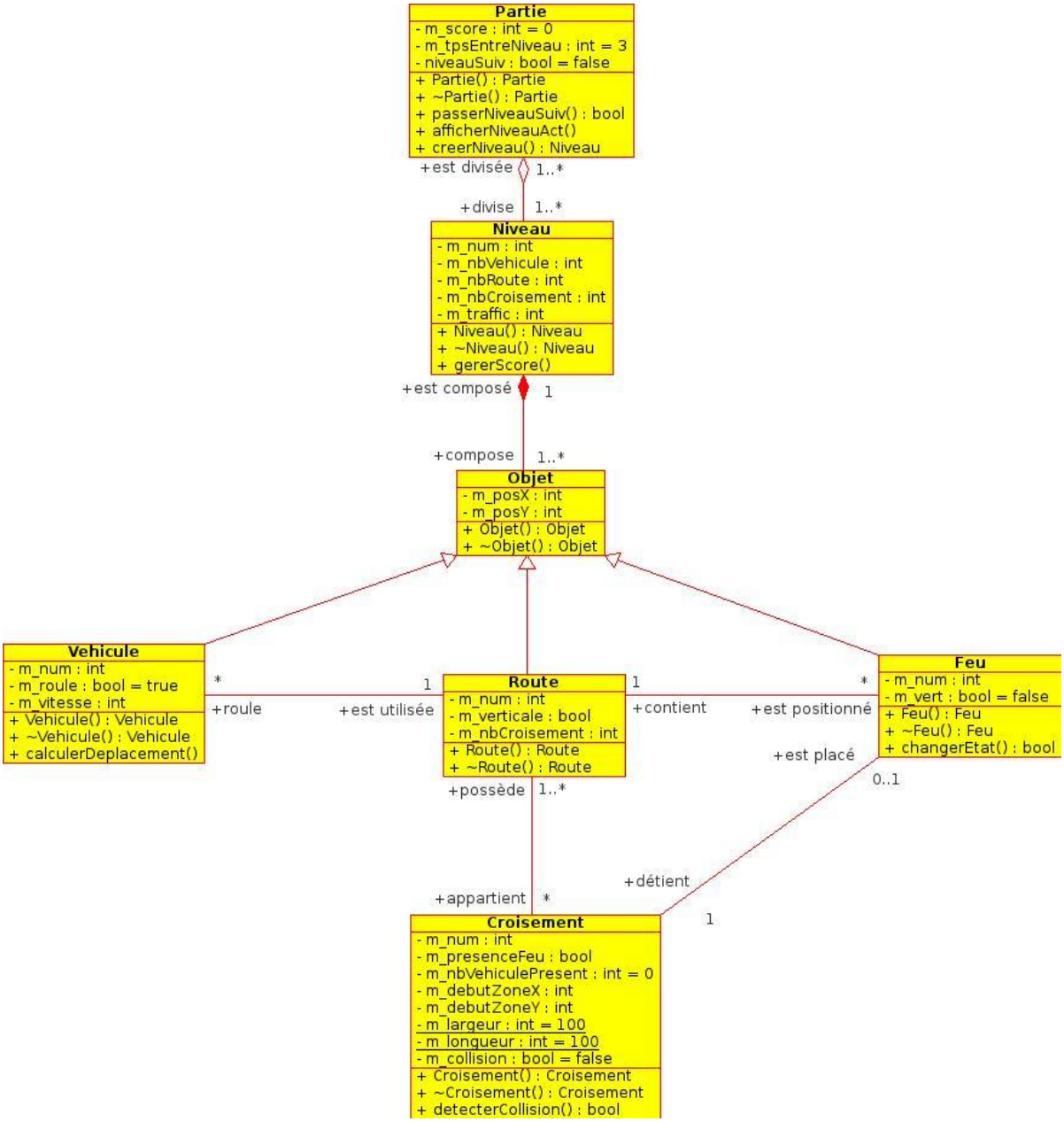
Ci-dessous la liste exhaustive des fonctionnalités que notre projet possède:

- Ecran d'introduction de jeu
- Ecran de menu permettant via des boutons :
  - De lancer le jeu
- Ecran de « jeu » permettant d'afficher :
  - Les véhicules du jeu
  - Les routes
  - Les feux de signalisation
  - Le score, le taux d'embouteillage et le nombre de points restants à gagner pour réussir le niveau
- Ecran de « jeu » interactif permettant de changer la couleur des feux de signalisation
- Plusieurs niveaux.
- Ecran lors du chargement du niveau ou lorsque le niveau est échoué
- Ecran lors d'une défaite
- Editeur algorithmique de niveau

II] Le code

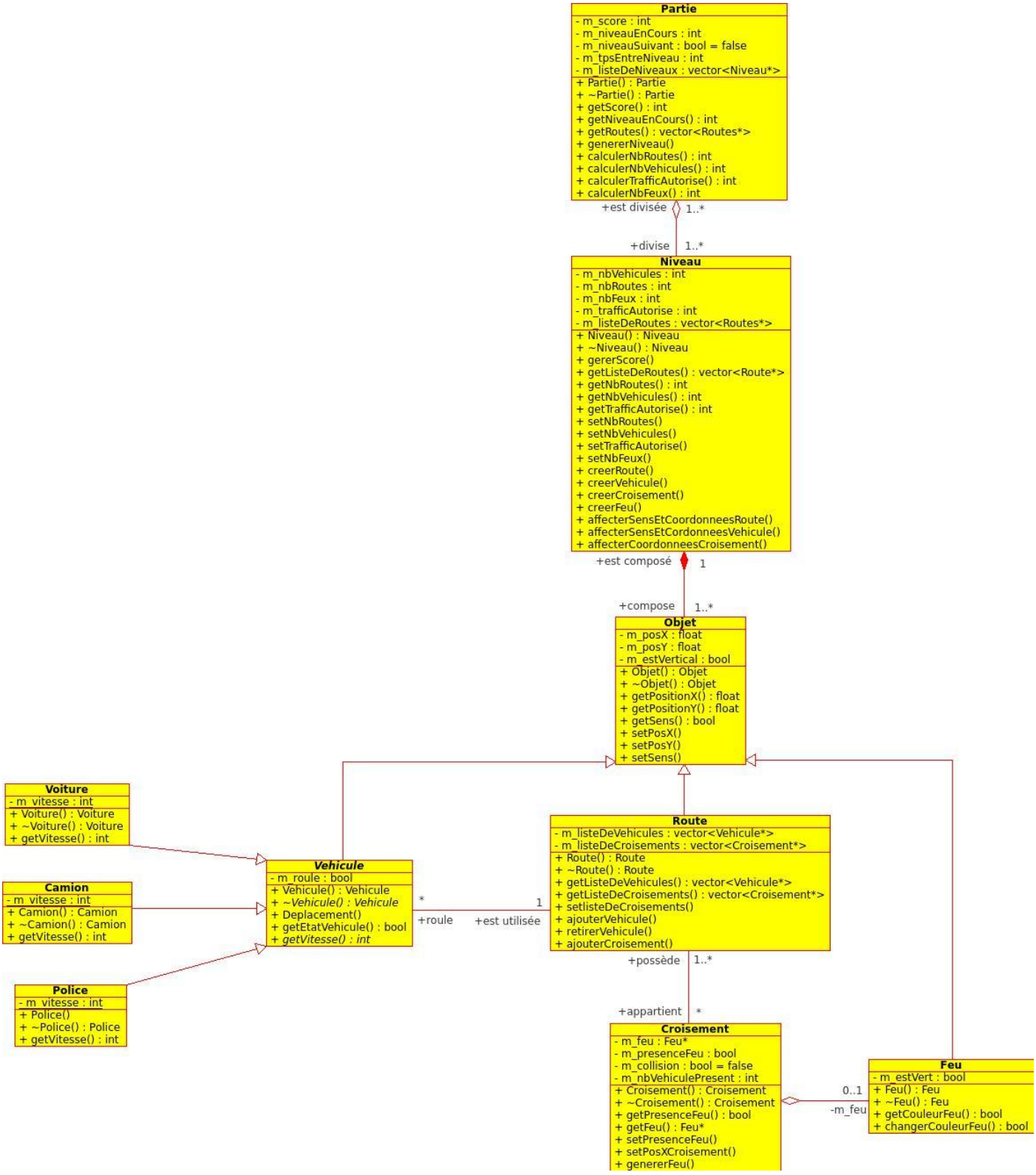
A. Architecture

Avant de commencer le code à proprement dire, nous avons tout d'abord réfléchi à une architecture possible de notre projet. Nous avons donc remis ce premier jet dans le rapport d'ACSI : Spécifications UML.



Dans cette première version, une partie était composée de plusieurs niveaux eux même composés de différents objets (la classe *Objet* était une classe abstraite). Les classe *Vehicule*, *Route* et *Feu* héritées des attributs et des méthodes de la classe *Objet*. La classe *Vehicule* possédait un attribut *m_vitesse* de type entier qui selon sa valeur (1, 2 ou 3) indiquait si le véhicule était une voiture, une voiture de police ou un camion. Nous avons également eu l'idée de créer une classe *Croisement* qui devait permettre de gérer plus facilement les collisions entre les véhicules, le placement des feux, et l'arrêt des véhicules au feu (si le feu est rouge). Chaque feu devait appartenir à une route et au croisement associé. Cependant, l'association entre la classe *Croisement* et la classe *Feu* était redondante.

Nous avons donc commencé le code en suivant cette architecture qui nous semblait correcte. Mais nous avons tout de suite vu les limites d'une telle conception. Si l'on voulait qu'un feu appartienne et à une route et à un croisement in nous fallait créer un attribut *m_feu* dans la classe *Route* et un pointeur de ce feu dans la classe *Croisement*. C'était redondant car un croisement appartenait déjà à une route. De plus, il nous est apparut plus simple de créer une classe pour chaque type de véhicule plutôt qu'une unique classe *Vehicule*. Nous avons donc repenser notre architecture.



Dans cette nouvelle architecture, nous avons modifié la classe *Vehicule* pour la rendre abstraite, nous avons créé les classes *Voiture*, *Camion* et *Police* qui possèdent chacune un attribut statique *m_vitesse* qui prend les valeurs 1,2 ou 3. De plus, un feu est contenu dans un croisement et plus dans une route.

B. Classes

Notre code comprend 12 classes :

- *GameModel*
- *GameView*
- *Partie*
- *Niveau*
- *Objet*
- *Route*
- *Croisement*
- *Feu*
- *Vehicule* (abstraite)
- *Voiture*
- *Camion*
- *Police*

Chaque classe possède des attributs et des méthodes qui lui sont propres que nous allons développés dans cette partie.

1. Attributs

Classe *GameModel* :

- *m_width* : int →Entier indiquant la largeur de la fenêtre d'affichage.
- *m_height* : int → Entier indiquant la hauteur de la fenêtre d'affichage.
- *nbVehicules* : int → Entier indiquant le nombre de véhicules qui doivent partir à chaque « top ». Il est égal au nombre de routes du niveau.
- *numeroVehicule* : int → Entier indiquant l'indice des véhicules à faire partir dans les vecteurs de véhicules de chaque route.
- *m_xCollision* : int → Entier indiquant la coordonnée sur l'axe des abscisses servant à l'affichage de l'animation de la collision.
- *m_yCollision* : int → Entier indiquant la coordonnée sur l'axe des abscisses servant à l'affichage de l'animation de la collision.
- *m_tempsCollision* : float → Nombre décimal indiquant le temps auquel la collision a eu lieu.
- *time* : float → Nombre décimal indiquant le temps d'attente entre chaque départ de véhicules.
- *m_collisionFinie* → Booléen indiquant si l'animation de la collision est finie (vrai) ou non (faux).
- *horloge* → Variable de type *Clock* qui stocke l'heure de l'ordinateur.
- *m_partie* → Pointeur vers une instance de la classe *Partie*.

Classe *GameView* :

- *m_width* : int →Entier indiquant la largeur de la fenêtre d'affichage.
- *m_height* : int → Entier indiquant la hauteur de la fenêtre d'affichage.
- *m_menu* : bool → Booléen indiquant s'il on est dans la boucle de menu (vrai) ou s'il on est dans la boucle de jeu (faux).

- *m_animation* : bool → Booléen permettant s'il est vrai et si on est dans le menu *(m_menu*==true) d'afficher l'écran de chargement avant le menu ou s'il est vrai et si on est dans le jeu (*m_menu*==false) d'afficher l'écran de chargement entre les niveaux.
- *m_model* : GameModel* → Pointeur vers une instance de la classe *GameModel*.
- *m_window* : RenderWindow* → Pointeur vers une instance de la classe *RenderWindow.*
- *m_font* : Font → Variable de type *Font* qui permet l'affichage de la police de caractère.
- *horologe* : Clock → Variable de type *Clock* qui stocke l'heure de l'ordinateur.
- *m_chargement* : View → Variable de type *View* permettant le chargement de l'écran avant le menu.
-
- m_interNiveau : View → Variable de type *View* permettant le chargement de l'écran entre chaque niveau.
- Déclaration de toutes les images
- Déclaration de tous les sprites.

Classe *Partie* :

- *m_score* : int → Entier indiquant le score actuel d'un niveau.
- *m_niveauEnCours* : int → Entier indiquant le numéro du niveau en cours.
- *m_niveau* : Niveau* → Pointeur vers une instance de la classe *Niveau*.

Classe *Niveau* :

- *m_nbRoutes* : int → Entier indiquant le nombre de routes qu'un niveau possède, c'est aussi la taille du vecteur de routes *m_listeDeRoutes*.
- *m_nbVehicules* : int → Entier indiquant le nombre de véhicules que chaque route d'un niveau possède, c'est aussi la taille du vecteur de véhicules *m_listeDeVehicules* de la classe *Route*.
- *m_nbFeux* : int → Entier indiquant le nombre de feux qu'un niveau possède.
- *m_trafficAutorise* : int → Entier indiquant le nombre maximal de véhicules présents en même temps sur la fenêtre de jeu.
- *m_objectif* : int → Entier indiquant le nombre de véhicules à faire traverser afin de réussir le niveau.
- *m_nbVehiculesPasses* : int → Entier indiquant le nombre actuel de véhicules ayant traversés la fenêtre de jeu.
- *m_nbVehiculePresent* : int → Entier indiquant le nombre total de véhicules présents sur les routes horizontales.
- *m_collision* : bool → Booléen indiquant si il y a une collision (vrai) ou non (faux).
- *m_listeDeRoutes* : vector <Route*> → Vecteur de routes indiquant la liste de routes que possède chaque niveau.

Classe *Objet* :

- *m_posX* : float → Nombre décimal indiquant la coordonnée sur l'axe des abscisses.
- *m_posY* : float → Nombre décimal indiquant la coordonnée sur l'axe des ordonnées.
- *m_estVertical* : bool → Booléen indiquant si l'objet est dans le sens vertical (vrai) ou horizontal (faux).

Classe *Route* :

- *m_listeDeVehicules* : vector <Vehicule*> → Vecteur de véhicules indiquant la liste de véhicules que possède chaque route.
- *m_listeDeCroisements* : vector <Croisement*> → Vecteur de croisements indiquant la liste de croisements que possède chaque route.

Classe *Croisement* :

- *m_feu* : Feu* → Pointeur vers une instance de la classe feu.
- *m_presenceFeu* : bool → Booléen indiquant si un croisement possède un feu (vrai) ou non (faux).
- *m_nbVehiculePresent* : int → Entier indiquant le nombre de véhicules présent sur un croisement.

Classe *Feu* :

- *m_estVert* : bool → Booléen indiquant si un feu est vert (vrai) ou rouge (faux).

Classe *Vehicule* :

- *m_roule* : bool → Booléen indiquant si un véhicule (voiture, police ou camion) roule (vrai) ou non (faux).

Classe *Voiture* :

- *m_vitesse* : int → Entier indiquant la vitesse de déplacement du type de véhicule voiture.

Classe *Camion* :

- *m_vitesse* : int → Entier indiquant la vitesse de déplacement du type de véhicule camion.

Classe *Police* :

- *m_vitesse* : int → Entier indiquant la vitesse de déplacement du type de véhicule police.

2. Méthodes

Classe *GameModel* :

- *GameModel()* : GameModel → Constructeur par défaut.
- *GameModel(int width, int height)* : GameModel → Constructeur paramétré prenant en entrée la largeur et la hauteur de la fenêtre d'affichage.
- *~GameModel()* : GameModel → Destructeur.
- *getPartie()* const : Partie* → Accesseur en lecture qui retourne le pointeur la partie.
- *getTemps()* const : float → Accesseur en lecture qui retourne le temps stocké dans la variable *horloge* de la classe *GameModel*.
- *getTempsCollision()* const : float → Accesseur en lecture qui retourne le temps stocké dans la variable *m_tempsCollision*.
- *getXCollision()* const : int → Accesseur en lecture qui retourne l'entier contenu dans la variable *m_xCollision*.
- *getYCollision()* const : int → Accesseur en lecture qui retourne l'entier contenu dans la variable *m_yCollision*.

- *nextStep()* : void → Fonction dans laquelle le jeu est implémenté (Ex : départ de véhicules, calcul des vitesse de déplacement des vehicules, déplacement des véhicules, collision, génération d'un nouveau niveau, …)
- *collision()* : void → Sert à l'arret des véhicules aux feux (s'ils sont rouges), à ce que les véhicules ne se chevauchent pas et passer à vrai l'attribut *m_collision* si une collision est détectée.

Classe *GameView* :

- *GameView(int width, int height)* : GameView → Constructeur paramétré prenant en entrée la largeur et la hauteur de la fenêtre d'affichage.
- *~GameView()* : GameView → Destructeur.
- *getMenu(GameView * view)* const : bool → Accesseur en lecture qui retourne le booléen indiquant s'il on est dans le menu ou non.
- *convertInt(int number)* : string → Convertit un int en chaine de caractère et retourne cette valeur.
- *ecranChargement()* : void → Affiche à l'écran d'introduction avant le menu (Appelée dans *Draw()*).
- *ecranInterNiveau(bool gagne)* : void →Affiche à l'écran de chargement entre les niveaux (Appelée dans *Draw()*).
- *declarationImages()* : void →Sert à déclarée toutes les images, les redimensionnées et leurs affecter leurs positions initiales.
- *draw()* : void → Gere tout l'interface graphique du jeu.
- *affichageEcran()* : void →Affiche l'écran de jeu, les vehicules, les feux,
- *affichageCroisement(int i)* : void → Affiche à l'écran les croisements.
- *treatEvents()* : bool →Traite les différents évènements du clavier et de la souris (Ex : le clic sur les feux, quitte si on appuie sur échap, …)

Classe *Partie* :

- Partie() : Partie → Constructeur par défaut.
- ~Partie() : Partie →Destructeur.
- getScore() const : int → Accesseur en lecture qui retourne le score du joueur.
- getNiveauEnCours() const : int → Accesseur en lecture qui retourne le numéro du niveau en cours.
- getNiveau() const : Niveau* → Accesseur en lecture qui retourne le pointeur du niveau.
- getRoutes() const : vector<Route*> → Accesseur en lecture qui retourne le vecteur de routes retourner par la méthode *getListeDeRoutes()* de la classe *Niveau*.
- setScore(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* et l'affecte à la variable *m_score*.
- setNiveauEnCours(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* et l'affecte à la variable *m_niveauEnCours*.
- void genererNiveau() : void → Génère un nouveau niveau.
- supprimerNiveau() : void → Supprime un niveau.
- calculerNbRoutes() : int → Retourne un entier correspondant au nouveau nombre de routes du niveau à générer.
- calculerNbVehicules() : int → Retourne un entier correspondant au nouveau nombre de véhicules par route du niveau à générer.
- calculerTrafficAutorise() : int → Retourne un entier correspondant au nouveau trafic autorisé du niveau à générer.
- calculerNbFeux() : int → Retourne un entier correspondant au nouveau nombre de feux du niveau à générer.
- calculerScore(int i, int j) : int → Retourne un entier qui prend la valeur 1 ou 3 selon que le vehicule passé est un camion (3) ou une voiture (1) ou une voiture de police (1).

- calculerObjectif() : int → Retourne un entier correspondant au nouveau nombre de véhicules horizontaux à faire traverser pour réussir le niveau.

Classe *Niveau* :

- Niveau() : Niveau → Constructeur par défaut.
- Niveau(int nbRoutes, int nbVehicules, int trafficAutorise, int nbFeux, int objectif) : Niveau → Constructeur paramétré prenant en paramètre le nombre de routes du niveau, le nombre de véhicules, le trafic autorisé et le nombre de feux et le nombre de véhicules à faire traverser pour reussir le niveau.
- ~Niveau() : Niveau → Destructeur.
- getListeDeRoutes() const : vector<Route*> → Accesseur en lecture qui retourne le vecteur de routes du niveau.
- getNbRoutes() const : int → Accesseur en lecture qui retourne le nombre de routes du niveau.
- getNbVehicules() const : int → Accesseur en lecture qui retourne le nombre de véhicules par route du niveau.
- getTrafficAutorise() const : int → Accesseur en lecture qui retourne le trafic autorisé du niveau.
- getNbFeux() const : int → Accesseur en lecture qui retourne le nombre de feux du niveau.
- getObjectif() const : int → Accesseur en lecture qui retourne le nombre véhicules horizontaux à faire traverser pour réussir le niveau.
- getCollision() const : bool → Accesseur en lecture qui retourne si il y a une collision ou non.
- getNbVehiculesPasses() const : int → Accesseur en lecture qui retourne le nombre actuel de véhicules ayant traversés.
- getNbVehiculePresent() const : int → Accesseur en lecture qui retourne le nombre total de véhicules présents sur les routes horizontales.
- setNbRoutes(int nb) : int → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable m_*nbRoutes*.
- setNbVehicules(int nb) : int → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_nbVehicules*.
- setTrafficAutorise(int nb) : int → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_trafficAutorise*.
- setNbFeux(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_nbFeux*.
- setCollision(bool b) : void → Accesseur en écriture qui affecte la valeur booléenne contenu dans le paramètre *nb* à la variable *m_collision*.
- setNbVehiculesPasses(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_nbVehiculesPasses*.
- setObjectif(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_objectif*.
- setNbVehiculePresent(int nb) : void → Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nb* à la variable *m_nbVehiculePresent*.
- creerRoute() : void → Ajoute une nouvelle route dans le vecteur de routes du niveau.
- creerVehicule() : void →Crée un nouveau véhicule.
- creerCroisement(int nbRouteHorizontale, int nbRouteVerticale) : void → Crée un nouveau croisement.
- creerFeu(int nbRouteHorizontale, int nbRouteVerticale, int nbFeu) : void → Crée un nouveau feu.
- affecterSensEtCoordonneesRoute(int &nbRouteHorizontale, int &nbRouteVerticale) : void → Affecte un sens et des coordonnées à chaque route du niveau.
- affecterSensEtCoordonneesVehicules() : void → Affecte un sens et des coordonnées à chaque véhicule de chaque route du niveau.

- affecterCoordonneeCroisement() : void → Affecte des coordonnées à chaque croisement du niveau.

Classe *Objet* :

- Objet() : Objet → Constructeur par défaut.
- Objet(float posX, float posY) : Objet → Constructeur paramétré prenant en paramètre la coordonnée sur l'axe des abscisses et la coordonnée sur l'axe des ordonnées.
- Objet(float posX, float posY, bool estVertical) : Objet → Constructeur paramétré prenant en paramètre la coordonnée sur l'axe des abscisses, la coordonnée sur l'axe des ordonnées et le sens à affecter à l'objet.
- ~Objet() : Objet → Destructeur.
- getPositionX() const : float → Accesseur en lecture qui retourne la coordonnée sur l'axe des abscisses d'un objet.
- getPositionY() const : float → Accesseur en lecture qui retourne la coordonnée sur l'axe des ordonnées d'un objet.
- getSens() const : bool → Accesseur en lecture qui retourne le sens d'un objet.
- setPosX(float posX) : void → Accesseur en écriture qui affecte le nombre décimal contenu dans le paramètre *posX* à la variable *m_posX*.
- setPosY(float posY) : void → Accesseur en écriture qui affecte le nombre décimal contenu dans le paramètre *posY* à la variable *m_posY*.
- setSens(bool estVertical) : void → Accesseur en écriture qui affecte la valeur booléenne contenu dans le paramètre *estVertical* à la variable *m_estVertical*.

Classe *Route* :

- Route() : Route → Constructeur par défaut.
- ~Route() : Route → Destructeur.
- getListeDeVehicules() const : vector<Vehicule*> → Accesseur en lecture qui retourne le vecteur de véhicules d'une route.
- getListeDeCroisements() const : vector<Croisement*> → Accesseur en lecture qui retourne le vecteur de croisements d'une route.
- ajouterVehicule() : void → Ajoute un nouveau véhicule dans le vecteur de véhicules de la route.
- ajouterCroisement() : void → Ajoute un nouveau croisement dans le vecteur de croisements de la route.

Classe *Croisement* :

- Croisement() : Croisement → Constructeur par défaut.
- ~Croisement() : Croisement → Destructeur.
- getPresenceFeu() const : bool → Accesseur en lecture qui retourne vrai si un croisement possède un feux et faux sinon.
- getFeu()const : Feu* → Accesseur en lecture qui retourne le pointeur vers le feu d'un croisement.
- getNbVehiculePresent() const : int → Accesseur en lecture qui retourne le nombre de véhicules présents sur le croisement.
- setPresenceFeu(bool presenceFeu) : void → Accesseur en écriture qui affecte la valeur booléenne contenu dans le paramètre *presenceFeu* à la variable *m_presenceFeu*.
- setPosXCroisement(std::vector <Croisement*> listeDeCroisement, int posX, int i) : void → Accesseur en écriture qui remplace la coordonnée sur l'axe des abscisse du paramètre *listeDeCroisement* d'indice *i* dans le vecteur de croisements de la route par la valeur contenue dans le paramètre *posX*.

- setNbVehiculePresent(int nbVehicule) : void →Accesseur en écriture qui affecte l'entier contenu dans le paramètre *nbVehicule* à la variable *m_nbVehicules*.
- genererFeu() : void → Crée un nouveau feu.
- detecterCollision() : bool → Retourne vrai si il y a une collision et faux sinon.

Classe *Feu* :

- Feu() : Feu →Constructeur par défaut.
- ~Feu() : → Destructeur.
- getCouleurFeu()const : bool → Accesseur en lecture qui retourne vrai si le feu est vert et faux sinon.
- changerCouleurFeu() : void → Accesseur en écriture qui inverse la valeur booléenne contenue dans la variable *m_estVert*.

Classe *Vehicule* :

- Vehicule() : Vehicule → Constructeur par défaut.
- virtual ~Vehicule() : Vehicule → Destructeur virtuel.
- getEtatVehicule() const : bool → Accesseur en lecture qui retourne vrai si un véhicule roule et faux sinon.
- setEtatVehicule(bool etat) : void → Accesseur en écriture qui affecte la valeur booléenne contenue dans le paramètre *etat* à la variable *m_roule*.
- virtual int getVitesse() : int → Accesseur en lecture virtuel qui retourne l'entier retourner par la méthode *getVitesse()*.
- deplacement() : void → Calcule les nouvelles coordonnées des véhicules.

Classe *Voiture* :

- Voiture() : Voiture → Constructeur par défaut.
- ~Voiture() : Voiture → Destructeur.
- getVitesse() : int → Accesseur en lecture qui retourne la vitesse du véhicule (2).

Classe *Camion* :

- Camion() : Camion → Constructeur par défaut.
- ~Camion() : Camion → Destructeur.
- getVitesse() : int → Accesseur en lecture qui retourne la vitesse du véhicule (1).

Classe *Police* :

- Police() : Police → Constructeur par défaut.
- ~Police() : Police → Destructeur.
- getVitesse() : Police → Accesseur en lecture qui retourne la vitesse du véhicule (3).

C. Principaux algorithmes

Dans cette partie, nous allons commenter et décrire étape par étape l'un des principaux algorithmes que contient notre code : l'éditeur algorithmique de niveau.

```
//(Editeur algorithmique de niveau)
Niveau::Niveau(int nbRoutes, int nbVehicules, int trafficAutorise, int nbFeux, int objectif):
        m_nbRoutes(nbRoutes), m_nbVehicules(nbVehicules),  m_nbFeux(nbFeux),
        m_trafficAutorise(trafficAutorise),  m_collision(0), m_objectif(objectif),
        m_nbVehiculesPasses(0),      m_nbVehiculePresent(0)
```

```
{
    (1) int nbRouteHorizontale=1, nbRouteVerticale=1, nbFeu=0;
    (2) creerRoute();
    (3) affecterSensEtCoordonneesRoute(nbRouteHorizontale, nbRouteVerticale);
    (4) creerVehicule();
    (5) affecterSensEtCoordonneesVehicules();
    (6) creerCroisement(nbRouteHorizontale, nbRouteVerticale);
    (7) affecterCoordonneeCroisement();
    (8) creerFeu(nbRouteHorizontale, nbRouteVerticale, nbFeux);
}
```

Notre éditeur est implémenté dans le constructeur paramétré de la classe *Niveau* afin que lors de la création d'un niveau, cela crée toutes les instances de classes nécessaires.

Détails des paramètres :

Le constructeur paramétré prend en paramètre le nombre de routes du niveau, le nombre de véhicules par routes, le trafic autorisé (nombre maximal de véhicules présents sur les routes horizontales en même temps), le nombre de feux et l'objectif (nombre de véhicules horizontaux à faire traverser pour réussir le niveau). Il affecte ces valeurs aux variables respectives *m_nbRoutes*, *m_nbVehicules*, *m_trafficAutorise*, *m_nbFeux* et *m_objetcif* et initialise les variables *m_collision*, *m_nbVehiculesPasses* et *m_nbVehiculePresent* à 0 ; c'est-à-dire qu'au début de chaque niveau, il n'y a pas de collision, aucun véhicule horizontal n'a encore traversé et aucun véhicule n'est afficher à l'écran.

(1) Création de trois variables de type entier. Elles nous serviront de compteur dans l'algorithme. Les variables *nbRouteHorizontale* et *nbRouteVerticale* sont initialisées à 1 car nous nous en servons également pour le calcul des coordonnées des routes que nous préciserons un peu plus loin. Elle compte respectivement le nombre actuel de routes horizontales créées et le nombre actuel de routes verticales créées. La variable *nbFeu* est initialisée normalement à 0 et compte le nombre actuel de feux créés.

(2) Appel de la méthode *creerRoute()*

```
void Niveau:: creerRoute()
{
	for (int i=0; i<m_nbRoutes; i++)
m_listeDeRoutes.push_back(new Route());
}
```

Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, on rajoute une case au vecteur de routes *m_listeDeRoutes* et on crée une nouvelle route.

(3) Appel de la méthode *affecterSensEtCoordonneesRoute (nbRouteHorizontale, nbRouteVerticale)*

```
void Niveau:: affecterSensEtCoordonneesRoute(int &nbRouteHorizontale, int &nbRouteVerticale)
{
	for (int i=0; i<m_nbRoutes; i++)
	{
		if (i%2==1)
			m_listeDeRoutes[i]->setSens(1);
		switch (m_listeDeRoutes[i]->getSens())
		{
			case 0:
			{
				m_listeDeRoutes[i]->setPosY(nbRouteVerticale*ECART_VERTICAL_ROUTES);
				nbRouteHorizontale++;
				break;
			}
			case 1:
			{
				m_listeDeRoutes[i]->setPosX((nbRouteHorizontale-1)*ECART_HORIZONTAL_ROUTES);
				nbRouteVerticale++;
				break;
			}
			default : break ;
		}
	}
}
```

On parcoure le vecteur de routes *m_listeDeRoutes* et on avance d'une case dans le vecteur par tour de boucle. Pour chaque tour de boucle : si le compteur de tour de boucle *i* est impair alors on affecte un sens vertical à la route d'indice *i* dans le vecteur. Ensuite selon le sens cette route, si elle est horizontale (*case 0*), on modifie sa coordonnée sur l'axe des ordonnées qui prend la valeur de la hauteur de la fenêtre moins le nombre de routes verticales crée multiplié par une constante qui permet un écart raisonnable entre les routes horizontales et on incrémente de 1 le nombre de routes verticales créées. A l'inverse, si la route est verticale, on modifie sa coordonnée sur l'axe des abscisses qui prend la valeur de la largeur de la fenêtre moins le nombre de routes horizontales crée multiplié par une constante qui permet un écart raisonnable entre les routes verticales et on incrémente de 1 le nombre de routes verticales créées. C'est pour cela que l'initialisation à 1 des variables *nbRouteHorizontale* et *nbRouteVerticale* est importante car sinon ce calcul pour les coordonnées ne fonctionnerait pas (la première route horizontale et la première route verticale seraient collées aux bords droit et inférieur de la fenêtre.

(4) Appel de la méthode *creerVehicule()*

```
void Niveau:: creerVehicule()
{
	for (int i=0; i<m_nbRoutes; i++)

		for (int k=0; k<m_nbVehicules;k++)
			m_listeDeRoutes[i]->ajouterVehicule();
}
```

Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, on rajoute une case au vecteur de routes *m_listeDeVehicules* et on fait appel a la méthode *ajouterVehicule()* de la classe *Route*. Cette méthode crée aléatoirement une voiture, un camion ou une voiture de police.

(5) Appel de le méthode *affecterSensEtCoordonneesVehicules()*

```
void Niveau:: affecterSensEtCoordonneesVehicules()
{
	for (int i=0; i<m_nbRoutes; i++)

		for (int j=0; j<m_listeDeRoutes[i]->getListeDeVehicules().size();j++)
			switch (m_listeDeRoutes[i]->getSens())
			{
				case 0:
				{
					m_listeDeRoutes[i]->getListeDeVehicules()[j]->setSens(0);
					m_listeDeRoutes[i]->getListeDeVehicules()[j]-
>setPosX(m_listeDeRoutes[i]->getPositionX()-128);
					m_listeDeRoutes[i]->getListeDeVehicules()[j]-
>setPosY(m_listeDeRoutes[i]->getPositionY()+5);
					break;
				}
				case 1:
				{
					m_listeDeRoutes[i]->getListeDeVehicules()[j]->setSens(1);
					m_listeDeRoutes[i]->getListeDeVehicules()[j]-
>setPosX(m_listeDeRoutes[i]->getPositionX()+5);
					m_listeDeRoutes[i]->getListeDeVehicules()[j]-
>setPosY(m_listeDeRoutes[i]->getPositionY()-128);
					break;
				}
				default: break;
			}
}
```

Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, on parcoure le vecteur de véhicules *m_listeDeVehicules* de chaque route du niveau. Si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est horizontale (*case 0*) on affecte le sens horizontal à chaque véhicules du vecteur de véhicules *m_listeDeVehicules* de cette route et on affect les coordonnées de départ appropriées aux véhicules. A l'inverse, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est verticale (*case 1*) on affecte le sens vertical à chaque véhicules du vecteur de véhicules *m_listeDeVehicules* de cette route et on affect les coordonnées de départ appropriées aux véhicules.

(6) Appel de la méthode *creerCroisement (nbRouteHorizontale, nbRouteVerticale)*

```
void Niveau:: creerCroisement(int nbRouteHorizontale, int nbRouteVerticale)
{
	for (int i=0; i<m_nbRoutes; i++)
	{
		if (m_listeDeRoutes[i]->getSens()==false)
			for (int j=0; j<nbRouteVerticale-1; j++)
			{
				m_listeDeRoutes[i]->ajouterCroisement();
				m_listeDeRoutes[i]->getListeDeCroisements()[j]->setPosY(m_listeDeRoutes[i]-
>getPositionY());
			}
		else
			for (int j=0; j<nbRouteHorizontale-1; j++)
			{
				m_listeDeRoutes[i]->ajouterCroisement();
				m_listeDeRoutes[i]->getListeDeCroisements()[j]->setPosX(m_listeDeRoutes[i]-
>getPositionX());
			}
	}
}
```

Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est horizontale, pour chaque tour de boucle allant de 0 au nombre de routes verticales créées -1, on fait appel à la méthode *ajouterCroisement()* de la classe *Route* qui fait appel au constructeur par défaut de la classe *Croisement*. On affecte ensuite la coordonnée sur l'axe des ordonnées de la route d'indice i du vecteur de routes à la coordonnée sur l'axe des abscisses de ce croisement. A l'inverse, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est verticale, pour chaque tour de boucle allant de 0 au nombre de routes horizontales créées -1, on fait appel à la méthode *ajouterCroisement()*. On affecte ensuite la coordonnée sur l'axe des abscisses de la route d'indice i du vecteur de routes à la coordonnée sur l'axe des ordonnées de ce croisement.

(7) Appel à la méthode *affecterCoordonneeCroisement()*

```
void Niveau:: affecterCoordonneeCroisement()
{
	for (int i=0; i<m_nbRoutes; i++)
	{
		if (m_listeDeRoutes[i]->getSens() == false)
			for (int j=0 ; j <= m_listeDeRoutes[i]->getListeDeCroisements().size() ; j++)
				if (m_listeDeRoutes[j]->getSens()==true)
					for (int k=0; k<m_listeDeRoutes[i]->getListeDeCroisements().size();k++)

						m_listeDeRoutes[i]->getListeDeCroisements()[k]-
>setPosX((k+1)*ECART_HORIZONTAL_ROUTES);
		if (m_listeDeRoutes[i]->getSens() == true)
			for (int j=0 ; j <= m_listeDeRoutes[i]->getListeDeCroisements().size() ; j++)
				if (m_listeDeRoutes[j]->getSens()==false)
					for (int k=0; k<m_listeDeRoutes[i]->getListeDeCroisements().size();k++)

						m_listeDeRoutes[i]->getListeDeCroisements()[k]-
>setPosY((k+1)*ECART_VERTICAL_ROUTES);
	}
}
```

Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est horizontale, on parcoure le vecteur de croisements *m_listeDeCroisements* de la classe *Route* et si la route d'indice j (compteur de tour de boucle) est verticale alors on affecte la coordonnée sur l'axe des ordonnées qui corresponds à l'indice du croisement +1 multiplié par la constante d'écart entre les routes horizontales. A l'inverse, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est

verticale, on parcoure le vecteur de croisements *m_listeDeCroisements* de la classe *Route* et si la route d'indice j (compteur de tour de boucle) est horizontale alors on affecte la coordonnée sur l'axe des abscisses qui corresponds à l'indice du croisement +1 multiplié par la constante d'écart entre les routes verticales.

(8) Appel de la méthode *creerFeu()*

```
void Niveau:: creerFeu(int nbRouteHorizontale, int nbRouteVerticale, int nbFeux)
{
	int nb=0;
	for (int i=0; i<m_nbRoutes; i++)
	{
		if (m_listeDeRoutes[i]->getSens()==false)
			for (int j=0; j<nbRouteVerticale-1; j++)
			{
				if (nb<nbFeux)
				{
					m_listeDeRoutes[i]->getListeDeCroisements()[j]->setPresenceFeu(1);
					m_listeDeRoutes[i]->getListeDeCroisements()[j]->genererFeu();
					nb++;
				}
			}
	}
}
```

On crée un compteur de feux créés (variable *nb*). Pour chaque tour de boucle allant de 0 au nombre de routes du niveau -1, si la route d'indice *i* dans le vecteur de routes *m_listeDeRoutes* est horizontale, pour chaque tour de boucle allant de 0 au nombre de routes verticales créées -1, si le compteur de feux (*nb*) est strictement inférieur au nombre de feux du niveau alors on modifie le croisement pour qu'il puisse posséder un feu, et on fait appel à la méthode *genererFeu()* de la classe *Croisement* qui fait appel au constructeur par défaut de la classe *Feu*.

Maintenant que nous avons décris comment fonctionne cet éditeur algorithmique de niveau, nous allons voir comment il est appellé et comment sont calculer ses paramètres.

Appel du contructeur paramétré de niveau :

Ce constructeur est appellé dans le fichier *Partie.cc* et plus précisément dans la méthode *genererNiveau()*.

```
void Partie::genererNiveau()
{
	m_niveau=new Niveau(calculerNbRoutes(), calculerNbVehicules(), calculerTrafficAutorise() , calculerNbFeux(),
	calculerObjectif());
}
```

La méthode *calculerNbRoutes()* retourne un entier qui augmente de 1 par niveau si le niveau en cours est strictement inférieur à 4 ou qui prend la valeur du niveau en cours si le numéro du niveau en cours est compris entre 4 et le nombre de routes maximal du jeu (constante) ou le nombre maximal de routes dans le jeu dans tous les autres cas.

La méthode *calculerNbVehicules()* retourne un entier qui prend la valeur du niveau en cours +5.

La méthode *calculerNbTrafficAutorise()* retourne un entier qui prend la valeur de l'objectif à atteidre divisé par deux.

La méthode *calculerNbFeux()* retourne un entier qui prend la valeur du niveau en cours.

La méthode *calculerObjectif()* retourn un entier qui prend la valeur du niveau en cours multiplié par 5.

Conclusion

Ce projet tuteuré nous a permis de progresser en C++ ainsi que d'appliquer les compétences de Programmation Orientée Objet apprises en cours. Il nous à également permis de découvrir et d'utiliser les librairies STL et SFML. On à également développer notre capacité à travailler en équipe, qualité qui est indispensable à tout développeur. Le fait de mener à bien un projet en partant de zéro est aussi une expérience enrichissante puisqu'elle permet de mettre en relation les notions d'ACSI, d'UML et d'AP2.